Introduction

1.9 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

1.1 Errors in a computer program can be classified according to when they are detected and, if they are detected at compile time, what part of the compiler detects them. Using your favorite imperative language, give an example of each of the following.

(a) A lexical error, detected by the scanner
(b) A syntax error, detected by the parser
(c) A static semantic error, detected by semantic analysis
(d) A dynamic semantic error, detected by code generated by the compiler
(e) An error that the compiler can neither catch nor easily generate code to catch (this should be a violation of the language definition, not just a program bug)

Answer: There are many possible answers to this question. Here are possibilities in C:

Lexical error: '`@`' sign outside of a string or comment

Syntax error: mismatched parentheses in an arithmetic expression

Static semantic error: use of an identifier that was never declared

Dynamic semantic error: divide by zero

Error that can't reasonably be caught: failure to reclaim dynamically allocated objects that are no longer needed ("memory leak")

1.2 Consider again the Pascal tool set distributed by Niklaus Wirth (Example 1.15). After successfully building a machine language version of the Pascal compiler, one could in principle discard the P-code interpreter and the P-code version of the compiler. Why might one choose *not* to do so?

Answer: One obvious answer is that if the machine language version of the compiler were lost, or if one upgraded to a new machine architecture, one might need the tools to rebuild the compiler. Even if the compiler remains perfectly usable, however, it may be worthwhile to keep the tools around. The P-code version of a program tends to be significantly smaller than its machine language counterpart. On a circa 1970 machine, the savings in memory and disk requirements could really be important. Moreover, as noted in Section 1.4, an interpreter will often provide better run-time diagnostics than will the output of a compiler; these can be particularly valuable during program development. Finally, an interpreter allows a program to be rerun immediately after modification, without waiting for recompilation and linking—a feature that can also be handy for development. Some of the best programming environments for imperative languages (including most implementations of Java) include both a compiler and an interpreter.

1.3 Imperative languages like Fortran and C are typically compiled, while scripting languages, in which many issues cannot be settled until run time, are typically interpreted. Is interpretation simply what one "has to do" when compilation is infeasible, or are there actually some *advantages* to interpreting a language, even when a compiler is available?

Answer: Compiled code usually runs significantly faster than interpreted code, so programmers interested in performance tend to demand compilers. Compilation also catches some errors earlier, when they are easier to fix, rather than waiting to detect them until the program actually runs. Interpretation definitely has its advantages, however; it is not just a last resort. It is very handy during development because it allows a newly modified program to be executed without waiting for potentially time-consuming compilation and linking steps. It may be desirable on small machines because it takes less space, both in memory when running and on disk (because there are no executables). Interpretation facilitates higher quality error messages, because the interpreter has access to the full program source. It is much easier to bootstrap or port an interpreter than a compiler, so experimental language implementations are very often based on interpreters.

1.4 The `gcd` program of Example 1.20 might also be written

```
int main() {
    int i = getint(), j = getint();
    while (i != j) {
        if (i > j) i = i % j;
        else j = j % i;
    }
    putint(i);
}
```

Does this program compute the same result? If not, can you fix it? Under what circumstances would you expect one or the other to be faster?

Answer: The difference between the two programs is in the two assignment statements: `i := i - j` and `j := j - i`, versus `i := i % j` and `j := j % i`. Suppose `i > j`. Then `i % j == i - (j * (i / j))`, where the slash (`/`) indicates integer division. The computation `i % j` therefore comes close to accomplishing in one iteration of the loop what would happen over the course of `(i / j)` iterations of the original loop. The exception arises in the case where i is a multiple of j. In this case modular division produces a zero, after which the program aborts with a divide-by-zero error. One possible fix capitalizes on the observation that (for positive numbers) `i % j` is always smaller than `i`:

```
int main() {
    int large = getint(), small = getint();
    if (large < small) {
        int temp = small;
        small = large;
        large = temp;
    }
    while (small != 0) {
        int temp = small;
        small = large % small;
        large = temp;
    }
    putint(i);
}
```

If we observe that when `i < j`, `i % j = i`, then we can also employ the following simpler program, at the expense of one useless extra division when `i` is initially smaller than `j`:

```
int main () {
    int i, j, t;
    i = getint();  j = getint();
    while (i != 0) {
        t = i;
        i = j % i;
        j = t;
    }
    putint(j);
}
```

If `i` and `j` are about the same magnitude, the original (subtraction-based) program may be faster, because subtraction is faster than division on many machines. If `i` and `j` are of different magnitude, the `%`-based version is likely to be faster.

1.5 Expanding on Example 1.25, trace an interpretation of the `gcd` program on the inputs 12 and 8. Which syntax tree nodes are visited, in which order?

Answer:

program
:=
call
(3) call `getint`

(return)	
(return)	
(5)	
(return)	assign 12 to `i`
:=	
call	
(3)	call `getint`
(return)	
(return)	
(6)	
(return)	assign 8 to `j`
while	
≠	
(5)	
(return)	
(6)	
(return)	
(return)	true
if	
>	
(5)	
(return)	
(6)	
(return)	
(return)	true
:=	"then" branch
–	
(5)	
(return)	
(6)	
(return)	
(return)	assign 4 to `i`
(return)	
(return)	back to while node
≠	
(5)	
(return)	
(6)	
(return)	
(return)	true
if	
>	
(5)	
(return)	
(6)	
(return)	
(return)	false
:=	"else" branch
–	

```
(6)
(return)
(5)
(return)
(return)    assign 4 to j
(return)
(return)    back to while node
≠
(5)
(return)
(6)
(return)
(return)    false
call
(5)
(return)
(4)         call putint(4)
(return)
(return)
(return)
(return)
(return)    back to program node
```

1.6 Both interpretation and code generation can be performed by traversal of a syntax tree. Compare these two kinds of traversals. In what ways are they similar/different?

Answer: Both kinds of traversal are most easily written as a set of mutually recursive subroutines, with (to first approximation) one routine for each kind of syntax tree node. The code generation traversal will tend to visit each node once, to generate corresponding target code. The interpretation traversal may never visit certain nodes (if their part of the program isn't needed in this particular execution); other nodes it may visit many times (e.g., if they are part of a loop body or a subroutine that is called repeatedly).

1.7 In your local implementation of C, what is the limit on the size of integers? What happens in the event of arithmetic overflow? What are the implications of size limits on the portability of programs from one machine/compiler to another? How do the answers to these questions differ for Java? For Ada? For Pascal? For Scheme? (You may need to find a manual.)

Answer: The answer here is of course implementation-dependent. Using `gcc` on a 32-bit x86/Linux system, an `int` is 32 bits long, and has values from $-2^{31} \ldots 2^{31} - 1$. Arithmetic overflow happens silently—numbers wrap around. Size limits significantly impact the portability of C programs. There are systems in which an `int` is 16 bits, or 36, or 64. The language definition requires at least 16 bits.

Java is uniform across platforms: an `int` is always 32 bits, signed two's complement.

Ada allows considerable variation across platforms, but imposes a variety of constraints on range and behavior. Among other things, the built-in `Integer` type must span at least $-2^{15} + 1 \ldots 2^{15} - 1$.

Pascal says that integers run from `-maxint` to `maxint`, where maxint is an implementation-defined whole number. Implementations are free to pick any convenient number.

Scheme guarantees arbitrary arithmetic precision on all platforms. Depending on the size of underlying integers, programs on different platforms may vary in execution speed.

1.8 The Unix `make` utility allows the programmer to specify *dependences* among the separately compiled pieces of a program. If file *A* depends on file *B* and file *B* is modified, `make` deduces that *A* must be recompiled, in case any of the changes to *B* would affect the code produced for *A*. How accurate is this sort of dependence management? Under what circumstances will it lead to unnecessary work? Under what circumstances will it fail to recompile something that needs to be recompiled?

Answer: `Make` depends on file modification times, maintained by the operating system. Because it works at the granularity of files, it will force recompilation of everything that depends on file *F* whenever anything in *F*—even a comment—changes. It will also force recompilation if the date on the file changes for a spurious reason: e.g., due to compression, copying, etc. By the same token, if file *G* depends on *F*, and the date on *G* changes for a spurious reason, `make` may fail to recognize that recompilation is needed. Because `make` operates independently of the compiler, and has no knowledge of language semantics, it may also fail to perform needed recompilations if the programmer makes an error in describing inter-file dependences.

1.9 Why is it difficult to tell whether a program is correct? How do you go about finding bugs in your code? What kinds of bugs are revealed by testing? What kinds of bugs are not? (For more formal notions of program correctness, see the bibliographic notes at the end of Chapter 4.)

Answer: There are two principal obstacles to telling whether a program is correct. The one people think of most often is telling whether the program does what it is "supposed to do." Basic results from computability theory tell us that it is impossible in general to tell whether a given program computes a given function. Good software engineering practices therefore tend to emphasize techniques that develop a *particular* program from its specifications in a rigorous way that preserves some notion of correctness. Even so, the sheer complexity of modern systems makes it difficult to apply any provably correct methodology in a completely rigorous way. In the current state of the art, bugs appear to be inevitable.

Testing can help to uncover bugs, but in any non-trivial program the number of different execution paths, and the number of input patterns that may push the program through those paths, far exceeds the number that can be explored during any reasonable period of testing. A well-known maxim holds that "testing can reveal the presence of bugs, but never their absence." The best software companies generally employ well-paid teams of testers whose jobs are carefully structured to reward the discovery of bugs. In far too many companies, however, testing is left to the same people who are writing the code. These people tend to have "blind spots" regarding possible sources of bugs, as well as mixed emotions about finding bugs at all.

Unfortunately, even given a perfect program development methodology, or a perfect testing regime, programs would continue to fail in use, because of the second, less often considered obstacle to correctness: namely, figuring out what the program is "supposed to do" in the first place. The history of computing is filled with computer-related disasters caused not because a program failed to meet its requirements, but because a set of inputs arose that the people who wrote the requirements never anticipated.

2 Programming Language Syntax

2.7 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

2.1 Write regular expressions to capture the following.

(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.

Answer: For notational convenience, assume that the expression **not**(a_1, a_2, ..., a_n) is shorthand for (b_1 | b_2 | ... | b_m) , where the b_js are all the characters in the alphabet other than the a_is. Also assume that **nl** represents the newline character. Clearly the expression starts and ends with double quotes. In the middle it contains a Kleene-star sequence of single or escaped characters. A single character cannot be a quote, newline, or backslash. An escaped character can be anything other than a newline. The following captures these rules:

string ⟶ `"` (**not**(\, `"`, **nl**) | \ **not**(**nl**))* `"`

(b) Comments in Pascal. These are delimited by `(*` and `*)` or by `{` and `}`. They are not permitted to nest.

Answer: Again we assume the existence of a **not** operator.

Pascal_comment ⟶ `(*` (**not**(`*`) | `*` **not**()))* $*^+$ `)`

---

1 Questions © 2015, Morgan Kaufmann Publishers, Inc.; solutions © 2015, Michael L. Scott. This material may not be copied or distributed without written permission of the publisher and author.

(c) Numeric constants in C. These are octal, decimal, or hexadecimal integers, or decimal or hexadecimal floating-point values. An octal integer begins with `0`, and may contain only the digits `0`–`7`. A hexadecimal integer begins with `0x` or `0X`, and may contain the digits `0`–`9` and `a`/`A`–`f`/`F`. A decimal floating-point value has a fractional portion (beginning with a dot) or an exponent (beginning with `E` or `e`). Unlike a decimal integer, it is allowed to start with `0`. A hexadecimal floating-point value has an optional fractional portion and a mandatory exponent (beginning with `P` or `p`). In either decimal or hexadecimal, there may be digits to the left of the dot, the right of the dot, or both, and the exponent itself is given in decimal, with an optional leading `+` or `-` sign. An integer may end with an optional `U` or `u` (indicating "unsigned"), and/or `L` or `l` (indicating "long") or `LL` or `ll` (indicating "long long"). A floating-point value may end with an optional `F` or `f` (indicating "float"—single precision) or `L` or `l` (indicating "long"—double precision).

Answer:

C_constant ⟶ *int_const* | *fp_const*
int_const ⟶ (*oct_int* | *dec_int* | *hex_int*) *int_suffix*
oct_int ⟶ `0` *oct_digit* *
dec_int ⟶ *nonzero_digit* *dec_digit* *
hex_int ⟶ (`0x` | `0X`) *hex_digit* *hex_digit* *
oct_digit ⟶ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7`
nonzero_digit ⟶ `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`
dec_digit ⟶ `0` | *nonzero_digit*
hex_digit ⟶ *dec_digit* | `a` | `b` | `c` | `d` | `e` | `f` | `A` | `B` | `C` | `D` | `E` | `F`
int_suffix ⟶ ϵ | *u_suffix* (*l_suffix* | *ll_suffix* | ϵ)
| *l_suffix* (*u_suffix* | ϵ) | *ll_suffix* (*u_suffix* | ϵ)
u_suffix ⟶ `u` | `U`
l_suffix ⟶ `l` | `L`
ll_suffix ⟶ `ll` | `LL`
fp_const ⟶ (*dec_fp* | *hex_fp*) (`f` | `F` | `l` | `L` | ϵ)
dec_fp ⟶ *dec_digit* *dec_digit* * (`E` | `e`) *exponent*
| *dec_digit* * (`.` *dec_digit* | *dec_digit* `.`) *dec_digit* * ((`E` | `e`) *exponent* | ϵ)
hex_fp ⟶ (`0x` | `0X`) *hex_digit* * (`.` *hex_digit* | ϵ | *hex_digit* `.`) *hex_digit* *
(`P` | `p`) *exponent*
exponent ⟶ (`+` | `-` | ϵ) *dec_digit* *dec_digit* *

(d) Floating-point constants in Ada. These match the definition of *real* in Example 2.3, except that (1) a digit is required on both sides of the decimal point, (2) an underscore is permitted between digits, and (3) 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 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.

Answer:

Ada_int ⟶ *digit* ((`_` | ϵ) *digit*)*
extended_digit ⟶ *digit* | `a` | `b` | `c` | `d` | `e` | `f` | `A` | `B` | `C` | `D` | `E` | `F`
Ada_extended_int ⟶ *extended_digit* ((`_` | ϵ) *extended_digit*)*
Ada_FP_num ⟶ ((*Ada_int* ((`.` *Ada_int* | ϵ))
| (*Ada_int* `#` *Ada_extended_int*
((`.` *Ada_extended_int*) | ϵ) `#`))
(((`e` | `E`) (`+` | `-` | ϵ) *Ada_int*) | ϵ)

(e) 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-10 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.)

Answer: *digit*$^+$ `#`* (`.` `#`* | ϵ) | *digit** `.` *digit*$^+$ `#`*

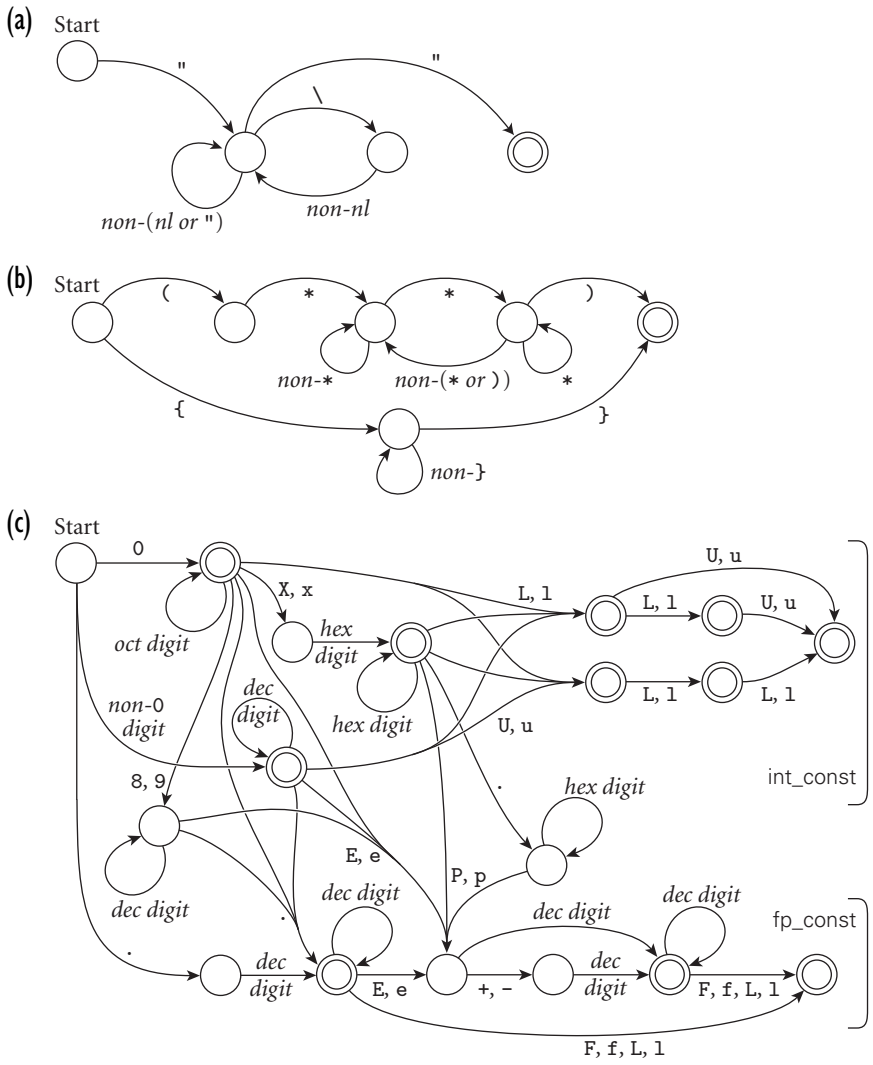
(f) Financial quantities in American notation. These have a leading dollar sign (`$`), an optional string of asterisks (`*`—used on checks to discourage fraud), a string of decimal digits, and an optional fractional part consisting of a decimal point (`.`) and two decimal digits. The string of digits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

Answer:

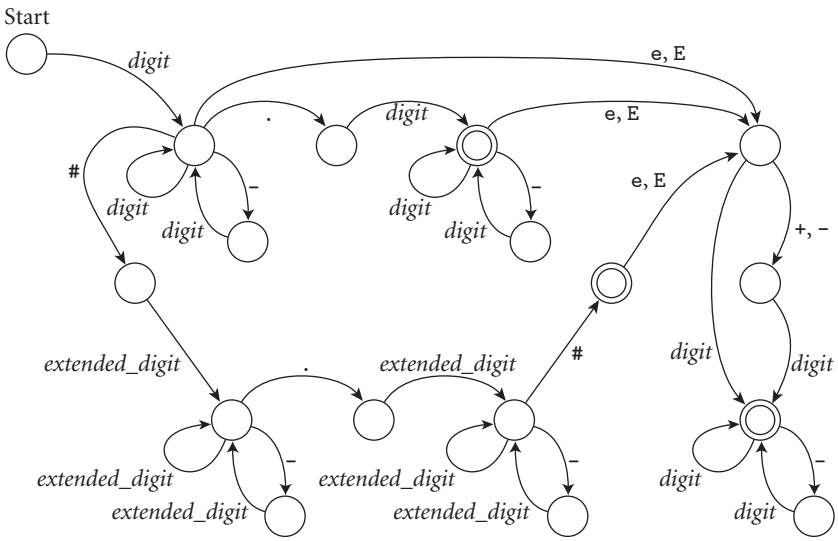
nzdigit ⟶ `1` | `2` | `3` | `4` | `5` | `4` | `7` | `8` | `9`
digit ⟶ `0` | *nzdigit*
group ⟶ `,` *digit digit digit*
number ⟶ `$*`* (`0` | *nzdigit* (ϵ | *digit* | *digit digit*) *group**) (ϵ | `.` *digit digit*)

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

Answer:


(a)
Start
"
"
\
non-(nl or ")
non-nl
(b)
Start
(
*
*
)
non-*
non-(* or))
*
{
}
non-}
(c)
Start
0
X, x
oct digit
hex digit
hex digit
L, l
L, l
U, u
U, u
L, l
L, l
non-0 digit
dec digit
U, u
int_const
8, 9
.
hex digit
E, e
P, p
dec digit
dec digit
dec digit
dec digit
fp_const
.
dec digit
E, e
+, –
dec digit
F, f, L, l
F, f, L, l

(d)



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.

Answer: In a manner similar to that of Exercise 2.1, we assume for notational convenience that the expression **not**($\mathtt{a}_1$, $\mathtt{a}_2$, ..., $\mathtt{a}_n$) is shorthand for ($\mathtt{b}_1$ | $\mathtt{b}_2$ | ... | $\mathtt{b}_m$) , where the $\mathtt{b}_j$s are all letters other than the $\mathtt{a}_i$s.[2] We also assume that *letter* stands for any letter. One way to generate a word other than `file`, `for`, or `from` is to start with something other than `f`:

non-keyword ⟶ **not**(`f`) *letter**

Another possibility is to follow `f` with something other than `i`, `o`, or `r`:

non-keyword ⟶ `f` **not**(`i`, `o`, `r`) *letter**

Yet another is to follow `fi` with something other than an `l`:

non-keyword ⟶ `fi` **not**(`l`) *letter**

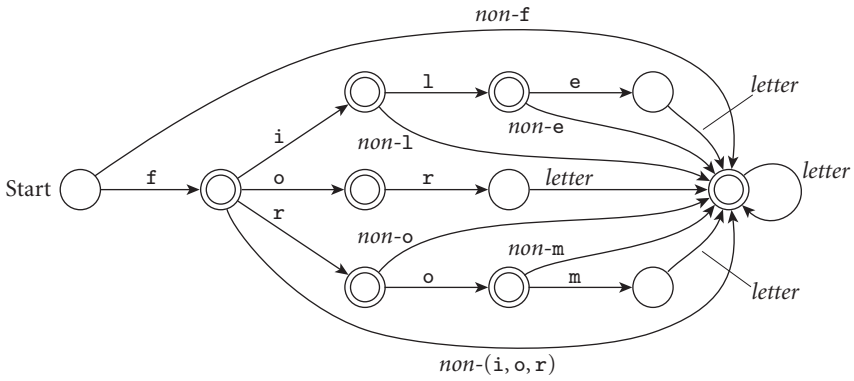
Fleshing out all such possibilities, we obtain

non-keyword ⟶ **not**(`f`) *letter** | `f` **not**(`i`, `o`, `r`) *letter**
| `fi` **not**(`l`) *letter** | `fo` **not**(`r`) *letter** | `fr` **not**(`o`) *letter**
| `fil` **not**(`e`) *letter** | `fro` **not**(`m`) *letter**
| `file` *letter*$^+$ | `for` *letter*$^+$ | `from` *letter*$^+$

---

2 This convention is not exactly the same as the one in Exercise 2.1. The **not** operator implicitly assumes an alphabet with respect to which to form the complement of its operand list. In this Example the alphabet is the set of letters; in Exercise 2.1 it was the set of printable characters.

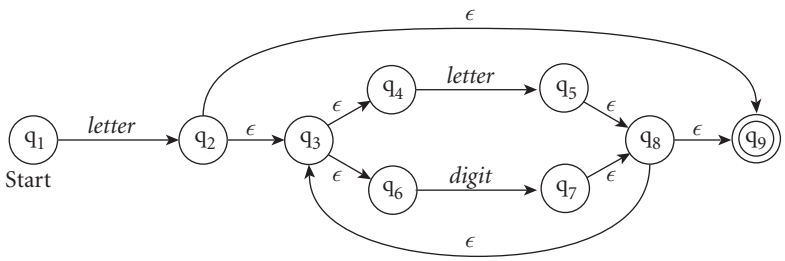
The corresponding finite automaton (not asked for in the question) is straightforward but tedious:



Our regular expression and automaton are clearly quite complex, with only three keywords to be excluded. For the 35 keywords of Pascal, the 50 of Java, or the 69 of Ada, they would be truly horrific.

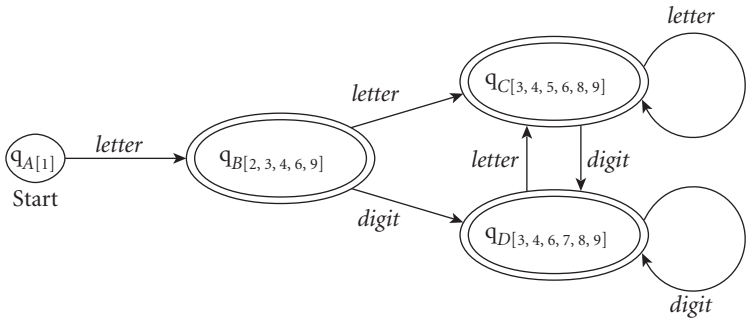
2.4 (a) Show the NFA that results from applying the construction of Figure 2.7 to the regular expression *letter* (*letter* | *digit*)*.

Answer:



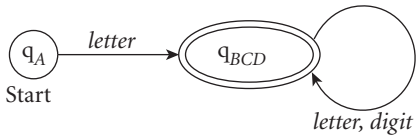
(b) Apply the transformation illustrated by Example 2.14 to create an equivalent DFA.

Answer:



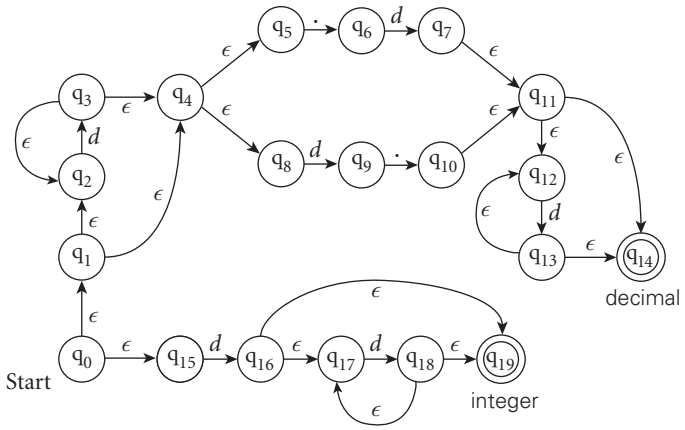
(c) Apply the transformation illustrated by Example 2.15 to minimize the DFA.

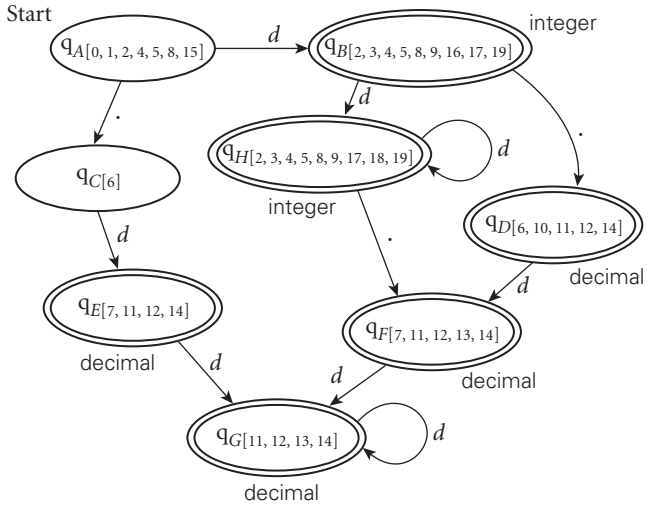
Answer:

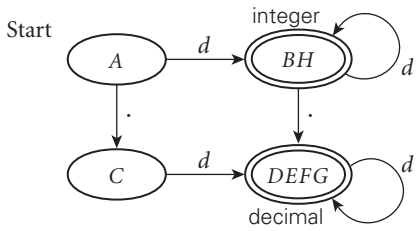


2.5 Starting with the regular expressions for *integer* and *decimal* in Example 2.3, construct an equivalent NFA, the set-of-subsets DFA, and the minimal equivalent DFA. Be sure to keep separate the final states for the two different kinds of token (see Sidebar 2.4). You may find the exercise easier if you undertake it by modifying the machines in Examples 2.13 through 2.15.

Answer:







2.6 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.

Answer: Here's a solution in C:

```
#include "stdio.h"
#include "ctype.h"
char token_image[100];    /* assume no token will be longer than this */
typedef enum {read, write, id, literal, becomes,
               add, sub, mul, div, lparen, rparen, eof} token;
char* names[] = {"read", "write", "id", "literal", "becomes",
               "add", "sub", "mul", "div", "lparen", "rparen", "eof"};
token scan() {
    static int c = ' ';
        /* next available char; extra (int) width accommodates EOF */
    int i = 0;              /* index into token_image */
    /* skip white space */
    while (isspace(c)) {
        c = getchar();
    }
    if (c == EOF)
        return eof;
    if (isalpha(c)) {
        do {
            token_image[i++] = c;
            c = getchar();
        } while (isalpha(c) || isdigit(c) || c == '_');
        token_image[i] = '\0';
        if (!strcmp(token_image, "read")) return read;
        else if (!strcmp(token_image, "write")) return write;
        else return id;
    }
    else if (isdigit(c)) {
        do {
            token_image[i++] = c;
            c = getchar();
        } while (isdigit(c));
        token_image[i] = '\0';
        return literal;
```

```
    } else switch (c) {
        case ':':
            if (c = getchar() != '=') {
                fprintf(stderr, "error\n");
                exit(1);
            } else {
                c = getchar();
                return becomes;
            }
            break;
        case '+': c = getchar(); return add;
        case '-': c = getchar(); return sub;
        case '*': c = getchar(); return mul;
        case '/': c = getchar(); return div;
        case '(': c = getchar(); return lparen;
        case ')': c = getchar(); return rparen;
        default:
            printf("error\n");
            exit(1);
    }
}
void main() {
    token t;
    do {
        t = scan();
        printf(names[t]);
        if (t == id || t == literal) printf(": %s", token_image);
        printf("\n");
    } while (t != eof);
}
```

2.7 Write a program in your favorite scripting language to remove comments from programs in the calculator language (Example 2.9).

Answer: Here's a solution in Perl:

```
#! /usr/bin/perl
# Remove comments from programs in the calculator language of Example 2.9.
# Uses % as the regexp delimiter to avoid quoting hassles with slashes.
OUTER: while (<>) {
    if (m%^([^/]|/[^/*])*//%) {
        # first comment in line is C++-style
        s%//.*$%%;    # delete end-of-line comments
    } else {
        s%/\*([^*]|\*[^/])*\*/%%g;      # delete middle-of-line comments
        if (s%/\*.*$%%) {               # multi-line comment
            print;
            while (<>) {
```

```
                if (s%^([^*]|\*[^/])*\*+/%%) {
                    redo OUTER;         # start over
                }
            }
        }
    }
    print;
}
```

2.8 Build a nested-case-statements finite automaton that converts all letters in its input to lower case, 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.

Answer: Here's a solution in C:

```
/* Map upper-case to lower, except within Pascal strings and comments */
#include <stdio.h>
main()
{
   int c;   /* char or EOF */
   enum {start, inquote, quote2, incomment1,
      Lcomment, Rcomment, incomment2} state;
   enum {false, true} mapping;
   state = start;  mapping = true;
   while ((c = getchar() ) != EOF) {
      switch (state) {
         case start:
            switch (c) {
               case '\'' :
                  state = inquote;
                  mapping = false;
                  break;
               case '{' :
                  state = incomment1;
                  mapping = false;
                  break;
               case '(' :
                  state = Lcomment;
                  break;
               default :
                  /* state remains start */
                  break;
            } break;
```

```
case inquote :
   switch (c) {
      case '\'' :
         state = quote2;
         break;
      default :
         /* state remains inquote */
         break;
   } break;
case quote2 :
   switch (c) {
      case '\'' :
         state = inquote;
         break;
      default :
         state = start;
         mapping = true;
         break;
   } break;
case incomment1 :
   switch (c) {
      case '}' :
         state = start;
         mapping = true;
         break;
      default : break;
         /* state remains incomment1 */
   } break;
case Lcomment :
   switch (c) {
      case '\'' :
         state = inquote;
         mapping = false;
         break;
      case '{' :
         state = incomment1;
         mapping = false;
         break;
      case '(' : break;
         /* state remains Lcomment */
      case '*' :
         state = incomment2;
         mapping = false;
         break;
      default :
         state = start;
         break;
   } break;
```

```
            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);
      }
   }
```

2.9 (a) Describe in English the language defined by the regular expression `a*( b a* b a* )*`. Your description should be a high-level characterization—one that would still make sense if we were using a different regular expression for the same language.

Answer: The set of all strings of `a`s and `b`s containing an even number of `b`s.

(b) Write an unambiguous context-free grammar that generates the same language.

Answer:

$G \longrightarrow$ *As* S

$S \longrightarrow$ `b` *As* `b` *As* S

$\longrightarrow$

As $\longrightarrow$ `a` *As*

$\longrightarrow$

Note that putting another *As* between the second `b` and the second S in the second production would make the grammar ambiguous.

(c) Using your grammar from part (b), give a canonical (right-most) derivation of the string `b a a b a a a b b`.

Answer:

$G \Longrightarrow$ *As* $\underline{S}$
$\Longrightarrow$ *As* b *As* b *As* $\underline{S}$
$\Longrightarrow$ *As* b *As* b *As* b *As* b *As* $\underline{S}$
$\Longrightarrow$ *As* b *As* b *As* b *As* b $\underline{As}$
$\Longrightarrow$ *As* b *As* b *As* b $\underline{As}$ b
$\Longrightarrow$ *As* b *As* b $\underline{As}$ b b
$\Longrightarrow$ *As* b *As* b a $\underline{As}$ b b
$\Longrightarrow$ *As* b *As* b a a $\underline{As}$ b b
$\Longrightarrow$ *As* b *As* b a a a $\underline{As}$ b b
$\Longrightarrow$ *As* b $\underline{As}$ b a a a b b
$\Longrightarrow$ *As* b a $\underline{As}$ b a a a b b
$\Longrightarrow$ *As* b a a $\underline{As}$ b a a a b b
$\Longrightarrow$ $\underline{As}$ b a a b a a a b b
$\Longrightarrow$ b a a b a a a b b

2.10 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.

2.11 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.)

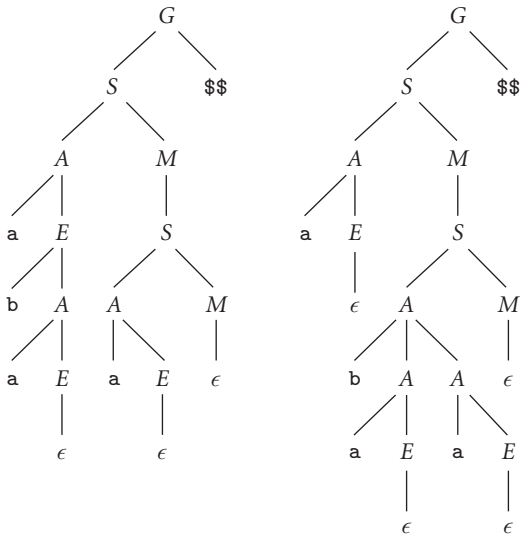
Answer: By definition, a grammar is LL(1) if it can be parsed by an LL(1) parser. It can be parsed by an LL(1) parser if no conflicts arise in the creation of the parse table. In this grammar, no symbols generate ϵ, so the table can be built entirely from FIRST sets; FOLLOW sets do not matter. There 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.

2.12 Consider the following grammar:

$G \longrightarrow S$ `$$`
$S \longrightarrow A\ M$
$M \longrightarrow S \mid \epsilon$
$A \longrightarrow$ `a` $E \mid$ `b` $A\ A$
$E \longrightarrow$ `a` $B \mid$ `b` $A \mid \epsilon$
$B \longrightarrow$ `b` $E \mid$ `a` $B\ B$

(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.

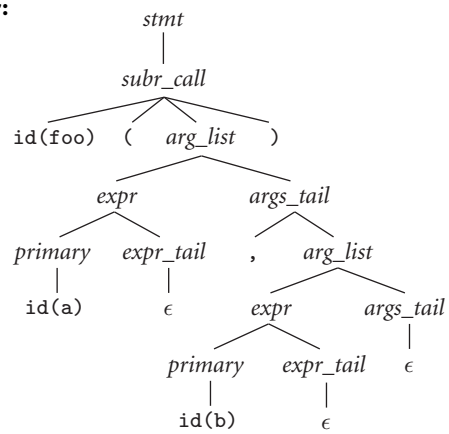
2.13 Consider the following grammar:

stmt $\longrightarrow$ *assignment*
$\longrightarrow$ *subr_call*
assignment $\longrightarrow$ `id :=` *expr*
subr_call $\longrightarrow$ `id (` *arg_list* `)`
expr $\longrightarrow$ *primary expr_tail*
expr_tail $\longrightarrow$ *op expr*
$\longrightarrow \epsilon$

primary ⟶ `id`
⟶ *subr_call*
⟶ `(` *expr* `)`
op ⟶ `+` | `-` | `*` | `/`
arg_list ⟶ *expr args_tail*
args_tail ⟶ `,` *arg_list*
⟶ ϵ

(a) Construct a parse tree for the input string `foo(a, b)`.

Answer:



(b) Give a canonical (right-most) derivation of this same string.

Answer:

stmt ⟹ *subr_call*
⟹ `id (` *arg_list* `)`
⟹ `id (` *expr args_tail* `)`
⟹ `id (` *expr* `,` *arg_list* `)`
⟹ `id (` *expr* `,` *expr args_tail* `)`
⟹ `id (` *expr* `,` *expr* `)`
⟹ `id (` *expr* `,` *primary expr_tail* `)`
⟹ `id (` *expr* `,` *primary* `)`
⟹ `id (` *expr* `, id )`
⟹ `id (` *primary expr_tail* `, id )`
⟹ `id (` *primary* `, id )`
⟹ `id ( id , id )`

(c) Prove that the grammar is not LL(1).

Answer: The token `id` is in both FIRST(*assignment*) and FIRST(*subr_call*), and thus in both PREDICT(*stmt* $\longrightarrow$ *assignment*) and PREDICT(*stmt* $\longrightarrow$ *subr_call*). Likewise, `id` is in both FIRST(`id`) and FIRST(*subr_call*), and thus in both PREDICT(*primary* $\longrightarrow$ `id`) and PREDICT(*primary* $\longrightarrow$ *subr_call*). Since the predict sets for both *stmt* and *primary* are not disjoint, the grammar is not LL(1).

(d) Modify the grammar so that it *is* LL(1).

Answer: Eliminate the *subr_call* production and replace the *stmt* and *primary* productions with the following (all else remains the same).

stmt $\longrightarrow$ `id` *stmt_tail*
stmt_tail $\longrightarrow$ `(` *arg_list* `)`
$\longrightarrow$ `:=` *expr*
primary $\longrightarrow$ `id` *primary_tail*
$\longrightarrow$ `(` *expr* `)`
primary_tail $\longrightarrow$ `(` *arg_list* `)`
$\longrightarrow$ ϵ

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

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

Answer:

LL(1) grammar:
1. $P \longrightarrow S$ `$$`
2. $S \longrightarrow$ `(` S `)` S
3. $S \longrightarrow$ `[` S `]` S
4. $S \longrightarrow \epsilon$

SLR(1) grammar:
1. $P \longrightarrow S$ `$$`
2. $S \longrightarrow S$ `(` S `)`
3. $S \longrightarrow S$ `[` S `]`
4. $S \longrightarrow \epsilon$

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

Answer:

LL(1) parse table:

Top-of-stack nonterminal	Current input token `(`	`)`	`[`	`]`	`$$`
P	1	–	1	–	1
S	2	4	3	4	4

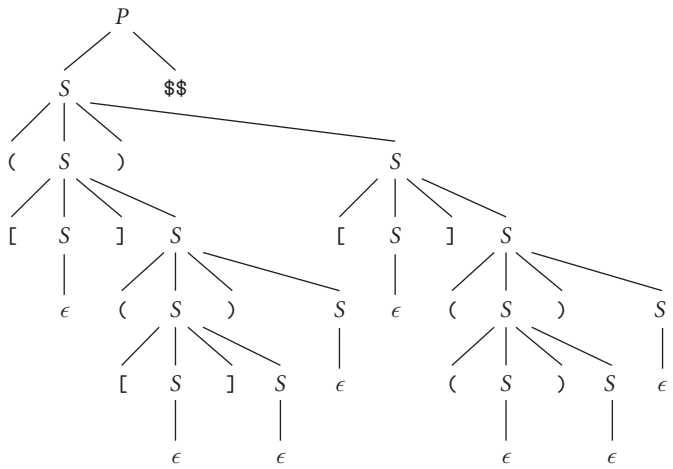
SLR(1) parse table:

Top-of-stack state	Current input symbol					
	S	(	)	[	]	$$
0	s1	r4	r4	r4	r4	r4
1	–	s2	–	s3	–	b1
2	s4	r4	r4	r4	r4	r4
3	s5	r4	r4	r4	r4	r4
4	–	s2	b2	s3	–	–
5	–	s2	–	s3	b3	–

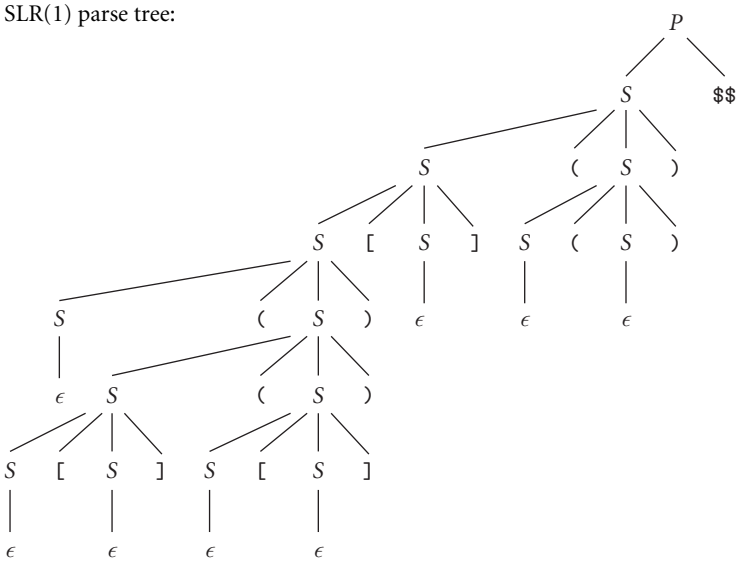
(c) For each grammar, show the parse tree for `([]([]))[](())`.

Answer:

LL(1) parse tree:



SLR(1) parse tree:



(d) Give a trace of the actions of the parsers in constructing these trees.

Answer:

LL(1) trace:

Parse stack	Input stream	Comment
P	`([]([]))[](()) $$`	
S `$$`	`([]([]))[](()) $$`	predict $P \longrightarrow S$ `$$`
`(` *S* `)` *S* `$$`	`([]([]))[](()) $$`	predict $S \longrightarrow$ `(` *S* `)` *S*
S `)` *S* `$$`	`[](([]))[](()) $$`	match `(`
`[` *S* `]` *S* `)` *S* `$$`	`[](([]))[](()) $$`	predict $S \longrightarrow$ `[` *S* `]` *S*
S `]` *S* `)` *S* `$$`	`](([]))[](()) $$`	match `[`
`]` *S* `)` *S* `$$`	`](([]))[](()) $$`	predict $S \longrightarrow \epsilon$
S `)` *S* `$$`	`(([]))[](()) $$`	match `]`
`(` *S* `)` *S* `)` *S* `$$`	`(([]))[](()) $$`	predict $S \longrightarrow$ `(` *S* `)` *S*
S `)` *S* `)` *S* `$$`	`([]))[](()) $$`	match `(`
`[` *S* `]` *S* `)` *S* `)` *S* `$$`	`([]))[](()) $$`	predict $S \longrightarrow$ `[` *S* `]` *S*
S `]` *S* `)` *S* `)` *S* `$$`	`]))[](()) $$`	match `[`
`]` *S* `)` *S* `)` *S* `$$`	`]))[](()) $$`	predict $S \longrightarrow \epsilon$
S `)` *S* `)` *S* `$$`	`))[](()) $$`	match `]`
`)` *S* `)` *S* `$$`	`))[](()) $$`	predict $S \longrightarrow \epsilon$
S `)` *S* `$$`	`)[](()) $$`	predict $S \longrightarrow \epsilon$
`)` *S* `$$`	`)[](()) $$`	predict $S \longrightarrow \epsilon$
S `$$`	`[](()) $$`	match `)`
`[` *S* `]` *S* `$$`	`[](()) $$`	predict $S \longrightarrow$ `[` *S* `]` *S*
S `]` *S* `$$`	`](()) $$`	match `[`
`]` *S* `$$`	`](()) $$`	predict $S \longrightarrow \epsilon$
S `$$`	`(()) $$`	match `]`

(*S*) *S* $$	(()) $$	predict $S \longrightarrow$ (*S*) *S*
S) *S* $$	()) $$	match (
(*S*) *S*) *S* $$	()) $$	predict $S \longrightarrow$ (*S*) *S*
S) *S*) *S* $$	)) $$	match (
) *S*) *S* $$	)) $$	predict $S \longrightarrow \epsilon$
S) *S* $$	) $$	match)
) *S* $$	) $$	predict $S \longrightarrow \epsilon$
S $$	$$	match)
$$	$$	predict $S \longrightarrow \epsilon$

SLR(1) trace:

Parse stack	Input stream	Comment
0	([]([]))[](()) $$	
0	*S* ([]([]))[](()) $$	reduce by $S \longrightarrow \epsilon$
0 S 1	([]([]))[](()) $$	shift *S*
0 S 1 (*2*	[]([]))[](()) $$	shift (
0 S 1 (*2*	*S* []([]))[](()) $$	reduce by $S \longrightarrow \epsilon$
0 S 1 (*2 S 4*	[]([]))[](()) $$	shift *S*
0 S 1 (*2 S 4* [*3*	]([]))[](()) $$	shift [
0 S 1 (*2 S 4* [*3*	*S*]([]))[](()) $$	reduce by $S \longrightarrow \epsilon$
0 S 1 (*2 S 4* [*3 S 5*	]([]))[](()) $$	shift *S*
0 S 1 (*2*	([]))[](()) $$	shift and reduce by $S \longrightarrow S$ [*S*]
0 S 1 (*2*	*S* ([]))[](()) $$	reduce by $S \longrightarrow \epsilon$
0 S 1 (*2 S 4*	([]))[](()) $$	shift *S*
0 S 1 (*2 S 4* (*2*	[]))[](()) $$	shift (
0 S 1 (*2 S 4* (*2*	*S* []))[](()) $$	reduce by $S \longrightarrow \epsilon$
0 S 1 (*2 S 4* (*2 S 4*	[]))[](()) $$	shift *S*
0 S 1 (*2 S 4* (*2 S 4* [*3*	]))[](()) $$	shift [
0 S 1 (*2 S 4* (*2 S 4* [*3*	*S*]))[](()) $$	reduce by $S \longrightarrow \epsilon$
0 S 1 (*2 S 4* (*2 S 4* [*3 S 5*	]))[](()) $$	shift *S*
0 S 1 (*2 S 4* (*2*	))[](()) $$	shift and reduce by $S \longrightarrow S$ [*S*]
0 S 1 (*2 S 4* (*2*	*S*))[](()) $$	reduce by $S \longrightarrow \epsilon$
0 S 1 (*2 S 4* (*2 S 4*	))[](()) $$	shift *S*
0 S 1 (*2*	)[](()) $$	shift and reduce by $S \longrightarrow S$ (*S*)
0 S 1 (*2*	*S*)[](()) $$	reduce by $S \longrightarrow \epsilon$
0 S 1 (*2 S 4*	)[](()) $$	shift *S*
0	[](()) $$	shift and reduce by $S \longrightarrow S$ (*S*)
0	*S* [](()) $$	reduce by $S \longrightarrow \epsilon$
0 S 1	[](()) $$	shift *S*
0 S 1 [*3*	](()) $$	shift [
0 S 1 [*3*	*S*](()) $$	reduce by $S \longrightarrow \epsilon$
0 S 1 [*3S5*	](()) $$	shift *S*
0	(()) $$	shift and reduce by $S \longrightarrow S$ [*S*]
0	*S* (()) $$	reduce by $S \longrightarrow \epsilon$
0 S 1	(()) $$	shift *S*
0 S 1 (*2*	()) $$	shift (
0 S 1 (*2*	*S* ()) $$	reduce by $S \longrightarrow \epsilon$

0 S 1 (*2 S 4*	()) $$	shift *S*
0 S 1 (*2 S 4* (*2*	)) $$	shift (
0 S 1 (*2 S 4* (*2*	*S*)) $$	reduce by $S \longrightarrow \epsilon$
0 S 1 (*2 S 4* (*2 S 4*	)) $$	shift *S*
0 S 1 (*2*	) $$	shift and reduce by $S \longrightarrow S$ (S)
0 S 1 (*2*	*S*) $$	reduce by $S \longrightarrow \epsilon$
0 S 1 (*2 S 4*	) $$	shift *S*
0	$$	shift and reduce by $S \longrightarrow S$ (S)
0	*S* $$	shift by $S \longrightarrow \epsilon$
0 S 1	$$	shift *S*
0	*P*	shift and reduce by $P \longrightarrow S$ $$
[done]		

2.15 Consider the following context-free grammar.

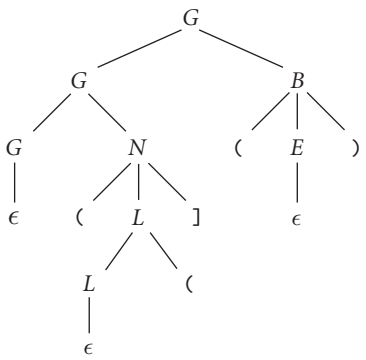
$$
\begin{aligned}
G &\longrightarrow G\ B \\
&\longrightarrow G\ N \\
&\longrightarrow \epsilon \\
B &\longrightarrow (\ E\ ) \\
E &\longrightarrow E\ (\ E\ ) \\
&\longrightarrow \epsilon \\
N &\longrightarrow (\ L\ ] \\
L &\longrightarrow L\ E \\
&\longrightarrow L\ (\\
&\longrightarrow \epsilon
\end{aligned}
$$

(a) Describe, in English, the language generated by this grammar. (Hint: *B* stands for "balanced"; *N* stands for "nonbalanced".) (Your description should be a high-level characterization of the language—one that is independent of the particular grammar chosen.)

Answer: Strings of balanced parentheses, with the exception that a right square bracket matches all still-open left parentheses that precede it. (Some dialects of Lisp permit these "close all" brackets.)

(b) Give a parse tree for the string ((](().

Answer:



(c) Give a canonical (right-most) derivation of this same string.

Answer:

$$
\begin{aligned}
G &\Longrightarrow G\ \underline{B} \\
&\Longrightarrow G\ (\ \underline{E}\ ) \\
&\Longrightarrow \underline{G}\ (\ ) \\
&\Longrightarrow G\ \underline{N}\ (\ ) \\
&\Longrightarrow G\ (\ \underline{L}\ ]\ (\ ) \\
&\Longrightarrow G\ (\ \underline{L}\ (\ ]\ (\ ) \\
&\Longrightarrow \underline{G}\ (\ (\ ]\ (\ ) \\
&\Longrightarrow (\ (\ ]\ (\ )
\end{aligned}
$$

(d) What is FIRST(E) in our grammar? What is FOLLOW(E)? (Recall that FIRST and FOLLOW sets are defined for symbols in an arbitrary CFG, regardless of parsing algorithm.)

Answer: FIRST(E) = {(}; FOLLOW(E) = {(,),]}

(e) Given its use of left recursion, our grammar is clearly not LL(1). Does this language have an LL(1) grammar? Explain.

Answer: No it doesn't. Proving this is difficult, but the intuition is straightforward: In an LL(1) parser, the contents of the parse stack represents the predicted remainder of the program, and decisions are always made on the basis of the top-of-stack symbol and the next token of input. When it matches a left parenthesis, the parser must predict whether it will see a matching right parenthesis, or whether this will be "swallowed up" by a right square bracket. Since the distance to the right parenthesis or bracket is unbounded, the parser cannot make this prediction with a fixed sized grammar. Note that eliminating left recursion does *not* suffice to make the grammar LL(1): there is potentially unbounded overlap between prefixes of the yields of B and N.

2.16 Give a grammar that captures all levels of precedence for arithmetic expressions in C, as shown in Figure 6.1. (Hint: This exercise is somewhat tedious. You'll probably want to attack it with a text editor rather than a pencil.)

Answer: The following is extracted from the ISO C standard [Int99, pp. 409–411]. It covers the full set of C operators, only some of which appear in Figure 6.1.

expression ⟶ *assignment_expression*
⟶ *expression* `,` *assignment_expression*
assignment_operator ⟶ `=` | `*=` | `/=` | `%=` | `+=` | `-=` | `<<=` | `>>=` | `&=` | `^=` | `|=`
assignment_expression ⟶ *conditional_expression*
⟶ *unary_expression assignment_operator assignment_expression*
conditional_expression ⟶ *logical_OR_expression*
⟶ *logical_OR_expression* `?` *expression* `:` *conditional_expression*
logical_OR_expression ⟶ *logical_AND_expression*
⟶ *logical_OR_expression* `||` *logical_AND_expression*
logical_AND_expression ⟶ *inclusive_OR_expression*
⟶ *logical_AND_expression* `&&` *inclusive_OR_expression*
inclusive_OR_expression ⟶ *exclusive_OR_expression*
⟶ *inclusive_OR_expression* `|` *exclusive_OR_expression*
exclusive_OR_expression ⟶ *AND_expression*
⟶ *exclusive_OR_expression* `^` *AND_expression*
AND_expression ⟶ *equality_expression*
⟶ *AND_expression* `&` *equality_expression*
equality_expression ⟶ *relational_expression*
⟶ *equality_expression* `==` *relational_expression*
⟶ *equality_expression* `!=` *relational_expression*
relational_expression ⟶ *shift_expression*
⟶ *relational_expression* `<` *shift_expression*
⟶ *relational_expression* `>` *shift_expression*
⟶ *relational_expression* `<=` *shift_expression*
⟶ *relational_expression* `>=` *shift_expression*
shift_expression ⟶ *additive_expression*
⟶ *shift_expression* `<<` *additive_expression*
⟶ *shift_expression* `>>` *additive_expression*
additive_expression ⟶ *multiplicative_expression*
⟶ *additive_expression* `+` *multiplicative_expression*
⟶ *additive_expression* `-` *multiplicative_expression*
multiplicative_expression ⟶ *cast_expression*
⟶ *multiplicative_expression* `*` *cast_expression*
⟶ *multiplicative_expression* `/` *cast_expression*
⟶ *multiplicative_expression* `%` *cast_expression*
cast_expression ⟶ *unary_expression*
⟶ `(` *type_name* `)` *cast_expression*

unary_operator ⟶ `&` | `*` | `+` | `-` | `~` | `!`
unary_expression ⟶ *postfix_expression*
⟶ `++` *unary_expression*
⟶ `--` *unary_expression*
⟶ *unary_operator cast_expression*
⟶ `sizeof` *unary_expression*
⟶ `sizeof (` *type_name* `)`
argument_expression_list ⟶ *assignment_expression*
⟶ *argument_expression_list* `,` *assignment_expression*
argument_expression_list_opt ⟶ *argument_expression_list*
⟶
postfix_expression ⟶ *primary_expression*
⟶ *postfix_expression* `[` *expression* `]`
⟶ *postfix_expression* `(` *argument_expression_list_opt* `)`
⟶ *postfix_expression* `.` *identifier*
⟶ *postfix_expression* `->` *identifier*
⟶ *postfix_expression* `++`
⟶ *postfix_expression* `--`
⟶ `(` *type_name* `) {` *initializer_list* `}`
⟶ `(` *type_name* `) {` *initializer_list* `, }`
primary_expression ⟶ *identifier*
⟶ *constant*
⟶ *string_literal*
⟶ `(` *expression* `)`

Syntax for *initializer_list*, *type_name*, *string_literal*, *constant*, and *identifier* is specified elsewhere in the standard.

2.17 Extend the grammar of Figure 2.25 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
```

Your grammar should support the six standard comparison operations in conditions, with arbitrary expressions as operands. It should also allow an arbitrary number of statements in the body of an `if` or `while` statement.

Answer: Add the following productions to Figure 2.25:

stmt ⟶ `if` *condition* `then` *stmt_list* `fi`
⟶ `while` *condition* `do` *stmt_list* `od`
condition ⟶ *expr relation expr*
relation ⟶ `<` | `>` | `<=` | `>=` | `=` | `!=`

2.18 Consider the following LL(1) grammar for a simplified subset of Lisp:

P ⟶ *E* `$$`
E ⟶ `atom`
⟶ `'` *E*
⟶ `(` *E Es* `)`
Es ⟶ *E Es*
⟶

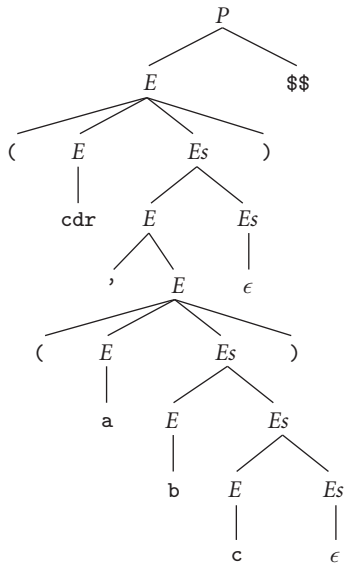
(a) What is FIRST(*Es*)? FOLLOW(*E*)? PREDICT(*Es* ⟶ ϵ)?

Answer:

{ `atom`, `(`, `'`, ϵ }
{ `atom`, `(`, `'`, `)`, `$$` }
{ `)` }

(b) Give a parse tree for the string `(cdr '(a b c)) $$`.

Answer:



(c) Show the left-most derivation of `(cdr '(a b c)) $$`.

Answer:

$P \Longrightarrow$ $\underline{E}$ `$$`
$\Longrightarrow$ `(` $\underline{E}$ *Es* `)` `$$`
$\Longrightarrow$ `(` `cdr` $\underline{Es}$ `)` `$$`
$\Longrightarrow$ `(` `cdr` $\underline{E}$ *Es* `)` `$$`
$\Longrightarrow$ `(` `cdr` `'` $\underline{E}$ *Es* `)` `$$`
$\Longrightarrow$ `(` `cdr` `'` `(` $\underline{E}$ *Es* `)` *Es* `)` `$$`
$\Longrightarrow$ `(` `cdr` `'` `(` `a` $\underline{Es}$ `)` *Es* `)` `$$`
$\Longrightarrow$ `(` `cdr` `'` `(` `a` $\underline{E}$ *Es* `)` *Es* `)` `$$`
$\Longrightarrow$ `(` `cdr` `'` `(` `a` `b` $\underline{Es}$ `)` *Es* `)` `$$`
$\Longrightarrow$ `(` `cdr` `'` `(` `a` `b` $\underline{E}$ *Es* `)` *Es* `)` `$$`
$\Longrightarrow$ `(` `cdr` `'` `(` `a` `b` `c` $\underline{Es}$ `)` *Es* `)` `$$`
$\Longrightarrow$ `(` `cdr` `'` `(` `a` `b` `c` `)` $\underline{Es}$ `)` `$$`
$\Longrightarrow$ `(` `cdr` `'` `(` `a` `b` `c` `)` `)` `$$`

(d) Show a trace, in the style of Figure 2.21, of a table-driven top-down parse of this same input.

Answer:

Parse stack	Input stream	Comment
P	`(cdr '(a b c)) $$`	
E `$$`	`(cdr '(a b c)) $$`	predict $P \longrightarrow E$ `$$`
`(` *E Es* `)` `$$`	`(cdr '(a b c)) $$`	predict $E \longrightarrow$ `(` *E Es* `)`
E Es `)` `$$`	`cdr '(a b c)) $$`	match `(`
`atom` *Es* `)` `$$`	`cdr '(a b c)) $$`	predict $E \longrightarrow$ `atom`
Es `)` `$$`	`'(a b c)) $$`	match `atom`
E Es `)` `$$`	`'(a b c)) $$`	predict $Es \longrightarrow E\ Es$
`'` *E Es* `)` `$$`	`'(a b c)) $$`	predict $E \longrightarrow$ `'` *E*
E Es `)` `$$`	`(a b c)) $$`	match `'`
`(` *E Es* `)` *Es* `)` `$$`	`(a b c)) $$`	predict $E \longrightarrow$ `(` *E Es* `)`
E Es `)` *Es* `)` `$$`	`a b c)) $$`	match `(`
`atom` *Es* `)` *Es* `)` `$$`	`a b c)) $$`	predict $E \longrightarrow$ `atom`
Es `)` *Es* `)` `$$`	`b c)) $$`	match `atom`
E Es `)` *Es* `)` `$$`	`b c)) $$`	predict $Es \longrightarrow E\ Es$
`atom` *Es* `)` *Es* `)` `$$`	`b c)) $$`	predict $E \longrightarrow$ `atom`
Es `)` *Es* `)` `$$`	`c)) $$`	match `atom`
E Es `)` *Es* `)` `$$`	`c)) $$`	predict $Es \longrightarrow E\ Es$
`atom` *Es* `)` *Es* `)` `$$`	`c)) $$`	predict $E \longrightarrow$ `atom`
Es `)` *Es* `)` `$$`	`)) $$`	match `atom`
`)` *Es* `)` `$$`	`)) $$`	predict $Es \longrightarrow \epsilon$
Es `)` `$$`	`) $$`	match `)`
`)` `$$`	`) $$`	predict $Es \longrightarrow \epsilon$
`$$`	`$$`	match `)`

(e) Now consider a recursive descent parser running on the same input. At the point where the quote token (') is matched, which recursive descent routines will be active (i.e., what routines will have a frame on the parser's run-time stack)?

Answer: P, E, Es, E, and match.

2.19 Write top-down and bottom-up grammars for the language consisting of all well-formed regular expressions. Arrange for all operators to be left-associative. Give Kleene closure the highest precedence and alternation the lowest precedence.

Answer: Top-down:

alternation ⟶ *concatenation more_concatenations*
more_concatenations ⟶ | *concatenation more_concatenations*
⟶
concatenation ⟶ *closure more_closures*
more_closures ⟶ *closure more_closures*
⟶
closure ⟶ *atom closure_option*
closure_option ⟶ *
⟶
atom ⟶ *character*
⟶ ϵ
⟶ (*alternation*)

Bottom-up:

alternation ⟶ *concatenation*
⟶ *alternation* | *concatenation*
concatenation ⟶ *closure*
⟶ *concatenation closure*
closure ⟶ *atom*
⟶ *atom* *
atom ⟶ *character*
⟶ ϵ
⟶ (*alternation*)

Note that in both of these grammars the vertical bar (|) and the ϵ symbol are part of the generated language, *not* part of the BNF notation.

2.20 Suppose that the expression grammar in Example 2.8 were to be used in conjunction with a scanner that did *not* remove comments from the input, but rather returned them as tokens. How would the grammar need to be modified to allow comments to appear at arbitrary places in the input?

Answer:

goal ⟶ *comment_opt expression comment_opt*
expression ⟶ *term* | *expression comment_opt add_op comment_opt term*
term ⟶ *factor* | *term comment_opt mult_op comment_opt factor*
factor ⟶ *identifier* | *number* | `-` *comment_opt factor* |
`(` *comment_opt expression comment_opt* `)`
add_op ⟶ `+` | `-`
mult_op ⟶ `*` | `/`
comment_opt ⟶ *comment* | ϵ

Clearly, applying this approach to a full-scale grammar is going to be very ugly.

2.21 Build a complete recursive descent parser for the calculator language. As output, have it print a trace of its matches and predictions.

Answer:

```
/* Complete recursive descent parser for the calculator language.
   Builds on Figure 2.17.  Uses the scanner from exercise 2.6.
   Does no error recovery.
*/

static token input_token;
void error() {
    printf("syntax error\n");
    exit(1);
}
void match(token expected) {
    if (input_token == expected) {
        printf("matched %s", names[input_token]);
        if (input_token == id || input_token == literal)
            printf(": %s", token_image);
        printf("\n");
        input_token = scan();
    }
    else error();
}
void program();
void stmt_list();
void stmt();
void expr();
void term_tail();
void term();
void factor_tail();
void factor();
void add_op();
void mult_op();
```

```
void program() {
    printf("predict program\n");
    switch (input_token) {
        case id:
        case read:
        case write:
        case eof:
            stmt_list();
            match(eof);
            break;
        default: error();
    }
}
void stmt_list() {
    printf("predict stmt_list\n");
    switch (input_token) {
        case id:
        case read:
        case write:
            stmt();
            stmt_list();
            break;
        case eof:
            break;          /*  epsilon production */
        default: error();
    }
}
void stmt() {
    printf("predict stmt\n");
    switch (input_token) {
        case id:
            match(id);
            match(becomes);
            expr();
            break;
        case read:
            match(read);
            match(id);
            break;
        case write:
            match(write);
            expr();
            break;
        default: error();
    }
}
```

```
void expr() {
    printf("predict expr\n");
    switch (input_token) {
        case id:
        case literal:
        case lparen:
            term();
            term_tail();
            break;
        default: error();
    }
}
void term_tail() {
    printf("predict term_tail\n");
    switch (input_token) {
        case add:
        case sub:
            add_op();
            term();
            term_tail();
            break;
        case rparen:
        case id:
        case read:
        case write:
        case eof:
            break;          /*  epsilon production */
        default: error();
    }
}
void term() {
    printf("predict term\n");
    switch (input_token) {
        case id:
        case literal:
        case lparen:
            factor();
            factor_tail();
            break;
        default: error();
    }
}
```

```
void factor_tail() {
    printf("predict factor_tail\n");
    switch (input_token) {
        case mul:
        case div:
            mult_op();
            factor();
            factor_tail();
            break;
        case add:
        case sub:
        case rparen:
        case id:
        case read:
        case write:
        case eof:
            break;          /*  epsilon production */
        default: error();
    }
}
void factor() {
    printf("predict factor\n");
    switch (input_token) {
        case id :
            match(id);
            break;
        case literal:
            match(literal);
            break;
        case lparen:
            match(lparen);
            expr();
            match(rparen);
            break;
        default: error();
    }
}
void add_op() {
    printf("predict add_op\n");
    switch (input_token) {
        case add:
            match(add);
            break;
```

```
        case sub:
            match(sub);
            break;
        default: error();
    }
}
void mult_op() {
    printf("predict mult_op\n");
    switch (input_token) {
        case mul:
            match(mul);
            break;
        case div:
            match(div);
            break;
        default: error();
    }
}
main() {
    input_token = scan();
    program();
}
```

2.22 Extend your solution to Exercise 2.21 to build an explicit parse tree.

2.23 Extend your solution to Exercise 2.21 to build an abstract syntax tree directly, without constructing a parse tree first.

2.24 The dangling `else` problem of Pascal was not shared by its predecessor Algol 60. To avoid ambiguity regarding which `then` is matched by an `else`, Algol 60 prohibited `if` statements immediately inside a `then` clause. The Pascal fragment

```
if C1 then if C2 then S1 else S2
```

had to be written as either

```
if C1 then begin if C2 then S1 end else S2
```

or

```
if C1 then begin if C2 then S1 else S2 end
```

in Algol 60. Show how to write a grammar for conditional statements that enforces this rule. (Hint: You will want to distinguish in your grammar between conditional statements and nonconditional statements; some contexts will accept either, some only the latter.)

Answer:

stmt $\longrightarrow$ *con_stmt* | *non_con_stmt*
con_stmt $\longrightarrow$ `if` *condition* `then` *non_con_stmt else_opt*
else_opt $\longrightarrow$ `else` *stmt* | ϵ
non_con_stmt $\longrightarrow$ `begin` *stmt_list* `end`
$\longrightarrow$ ...
stmt_list $\longrightarrow$ *stmt_list stmt* | ϵ
condition $\longrightarrow$ ...

2.25 Flesh out the details of an algorithm to eliminate left recursion and common prefixes in an arbitrary context-free grammar.

Answer: The following is based in part on Figures 3.4 and 3.6 (pages 96 and 101) in Cooper and Torczon's compiler text [CT04]. Equivalent treatment can be found in Algorithms 4.19 and 4.21 (pages 213 and 214) in the text of Aho et al. [ALSU07].

First we eliminate left recursion. Stated precisely, a grammar is left recursive if there is a nonterminal A such that $A \Longrightarrow^{+} A\ \alpha$. Assume for the moment that the grammar has no epsilon productions. We first choose an arbitrary order for the nonterminals: $A_1, A_2, \ldots, A_n$. We then back-substitute symbols into productions in such a way that, when we are done, the right hand side of a production for A_i never begins with an A_l for $l \leq i$. This "one way" property—the RHS of a production always begins with a terminal or a "later" nonterminal—ensures the absence of even indirect left recursion. Here's the algorithm:

for i in $1 \ldots n$
    -- Invariant: $\forall k<i$, $\nexists$ a production of the form $A_k \longrightarrow A_l\ \alpha$, for $l \leq k$
    for j in $1 \ldots i-1$
        for all productions P of the form $A_i \longrightarrow A_j\ \gamma$
            replace P with $A_i \longrightarrow \beta_1\gamma$ | $\beta_2\gamma$ | ... | $\beta_m\gamma$,
                where $A_j \longrightarrow \beta_1$ | β_2 | ... β_m are the current productions for A_j
    -- At this point A_i cannot be indirectly left recursive
    delete any production $A_i \longrightarrow A_i$
    if A_i is directly left recursive
        replace $A_i \longrightarrow A_i\ \delta_1$ | $A_i\ \delta_2$ | ... | $A_i\ \delta_s$ | ζ_1 | ζ_2 | ... | ζ_t
            with $A_i \longrightarrow \zeta_1\ B$ | $\zeta_2\ B$ | ... | $\zeta_t\ B$
            and $B \longrightarrow \delta_1\ B$ | $\delta_2\ B$ | ... | $\delta_s\ B$ | ϵ,
            where B is a new nonterminal

Epsilon productions in the original grammar cause trouble because they allow a "nullable" nonterminal at the beginning of a RHS to hide the left recursive nature of a following symbol. If our original grammar has epsilon productions we can get rid of them as follows:

-- Find all "nullable" symbols:
for all productions $A \longrightarrow \epsilon$, A is nullable
repeat
    for all productions $A \longrightarrow X_1\ X_2\ \ldots\ X_n$
        if $X_1, X_2, \ldots, X_n$ are all nullable, then A is nullable
until nothing new is learned

```
-- Build new grammar:
for all productions P = A ⟶ X1 X2 ... Xn, where n>0
    let R be {X1, X2, ..., Xn}
    let N ⊂ R be the nullable symbols in R
    for all M ⊂ N, M ≠ R
        place in the new grammar a variant of P
            in which all symbols in M have been removed from the RHS
if the start symbol S is nullable, place the following in the new grammar:
    T ⟶ S | ϵ, where T is a new nonterminal (and the new start symbol)
```

So now we can remove left recursion, direct or indirect, from any grammar. We can also remove direct left factors:

```
for each nonterminal A
    while there exist A productions with a nonempty common prefix
        let α be the longest nonempty prefix common to two or more A productions
        replace A ⟶ α β1 | α β2 | ... | α βn | γ1 | γ2 | ... | γm,
            where α is not a prefix of γi for any i
            with A ⟶ α B | γ1 | γ2 | ... | γm
            and B ⟶ β1 | β2 | ... | βn,
            where B is a new nonterminal
```

Note that this doesn't address the indirect case, where

$A \longrightarrow B\ \beta \mid C\ \gamma$
$B \longrightarrow \texttt{a}\ \delta$
$C \longrightarrow \texttt{a}\ \zeta$

Consider, for example, the language $\texttt{a}^n\texttt{b}^n \mid \texttt{a}^n\texttt{c}^n$. Here is an LR(0) grammar:

$S \longrightarrow B$
$\quad \longrightarrow C$
$B \longrightarrow \texttt{a}\ B\ \texttt{b}$
$\quad \longrightarrow \texttt{a}\ \texttt{b}$
$C \longrightarrow \texttt{a}\ C\ \texttt{c}$
$\quad \longrightarrow \texttt{a}\ \texttt{c}$

If we apply the left factoring algorithm we get

$S \longrightarrow B$
$\quad \longrightarrow C$
$B \longrightarrow \texttt{a}\ D$
$D \longrightarrow B\ \texttt{b}$
$\quad \longrightarrow \texttt{b}$
$C \longrightarrow \texttt{a}\ E$
$E \longrightarrow C\ \texttt{c}$
$\quad \longrightarrow \texttt{c}$

which still can't be parsed predictively. In fact, as noted in Figure C-2.39, this language has no LL grammar.

2.26 In some languages an assignment can appear in any context in which an expression is expected: the value of the expression is the right-hand side of the assignment, which is placed into the left-hand side as a side effect. Consider the following grammar fragment for such a language. Explain why it is not LL(1), and discuss what might be done to make it so.

expr ⟶ `id :=` *expr*
⟶ *term term_tail*
term_tail ⟶ `+` *term term_tail* | ϵ
term ⟶ *factor factor_tail*
factor_tail ⟶ `*` *factor factor_tail* | ϵ
factor ⟶ `(` *expr* `)` | `id`

Answer: The grammar is not LL(1) because `id` $\in$ PREDICT(*expr* ⟶ `id :=` *expr*) $\cap$ PREDICT(*expr* ⟶ *term term_tail*). It's trivially in the PREDICT set of the former production; it's in the PREDICT set of the latter production because *term* $\Longrightarrow^*$ *factor* ... $\Longrightarrow^*$ `id` .... To make it LL(1) we need to factor out the `id`. The easiest way to do this is to enlarge the language generated by the grammar:

expr ⟶ `term` *expr_tail*
expr_tail ⟶ `:=` *expr*
⟶ *term_tail*
term_tail ⟶ `+` *term term_tail* | ϵ
term ⟶ *factor factor_tail*
factor_tail ⟶ `*` *factor factor_tail* | ϵ
factor ⟶ `(` *expr* `)` | `id`

Semantic checks would then be required to ensure that the fragment to the left of a `:=` sign is an `id` and not some more complex sort of *term*.

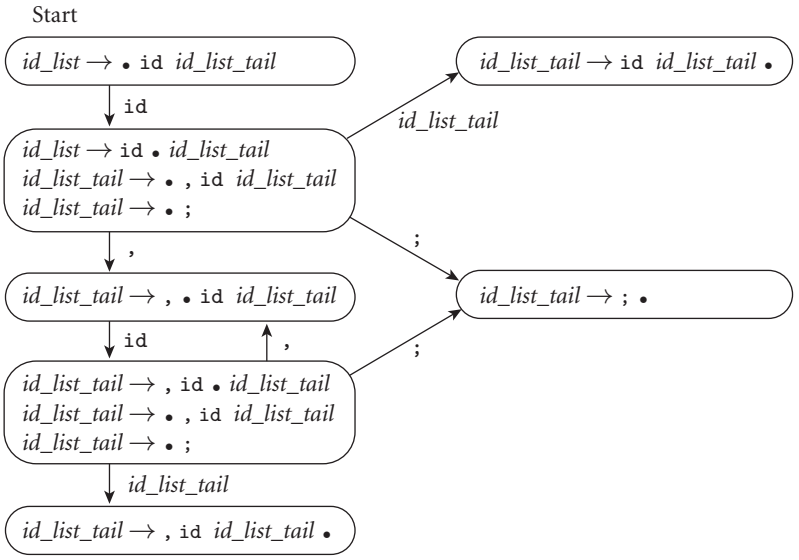
A precise solution that does not change the language is somewhat messier:

expr ⟶ `id` *id_expr_tail*
⟶ `(` *expr* `)` *factor_tail term_tail*
id_expr_tail ⟶ `:=` *expr*
⟶ *factor_tail term_tail*
term_tail ⟶ `+` *term term_tail* | ϵ
term ⟶ *factor factor_tail*
factor_tail ⟶ `*` *factor factor_tail* | ϵ
factor ⟶ `(` *expr* `)` | `id`

This solution becomes even messier if there are additional forms of *term* (e.g., function calls) that are acceptable as an rvalue but not as an lvalue.

2.27 Construct the CFSM for the *id_list* grammar in Example 2.20 and verify that it can be parsed bottom-up with *zero* tokens of look-ahead.

Answer:



Note that a dot appears at the end of an item in only three states, and there are no other items in these states. Lookahead is never required to resolve a shift-reduce conflict; the grammar is thus LR(0).

2.28 Modify the grammar in Exercise 2.27 to allow an *id_list* to be empty. Is the grammar still LR(0)?

Answer: It depends whether "empty" means the empty string or a bare semicolon. The grammar at left is LR(0); the grammar at right is not.

id_list ⟶ `id` *id_list_tail* | `;`
id_list_tail ⟶ `,` `id` *id_list_tail* | `;`

id_list ⟶ `id` *id_list_tail* | ϵ
id_list_tail ⟶ `,` `id` *id_list_tail* | `;`

2.29 Repeat Example 2.36 using the grammar of Figure 2.15.

2.30 Consider the following grammar for a declaration list:

decl_list ⟶ *decl_list decl* `;` | *decl* `;`
decl ⟶ `id` `:` *type*
type ⟶ `int` | `real` | `char`
⟶ `array const .. const of` *type*
⟶ `record` *decl_list* `end`

Construct the CFSM for this grammar.

Answer: The following CFSM is written in the style of Figure 2.26, with "shift and reduce" transitions used to eliminate trivial states. Given the existence of a terminating semicolon, we have not added the end-of-file marker `$$`.

	State	Transitions
0.	*decl_list* ⟶ • *decl_list decl* ; *decl_list* ⟶ • *decl* ; ——————— *decl* ⟶ • `id` : *type*	on *decl_list* shift and goto 1 on decl shift and goto 13 on `id` shift and goto 3
1.	*decl_list* ⟶ *decl_list* • *decl* ; ——————— *decl* ⟶ • `id` : *type*	on *decl* shift and goto 2 on `id` shift and goto 3
2.	*decl_list* ⟶ *decl_list decl* • ;	on ; shift and reduce (pop 2, push *decl_list*)
3.	*decl* ⟶ `id` • : *type*	on : shift and goto 4
4.	*decl* ⟶ `id` : • *type* ——————— *type* ⟶ • `int` | `real` | `char` *type* ⟶ • `array const .. const of` *type* *type* ⟶ • `record` *decl_list* `end`	on *type* shift and reduce (pop 2, push *decl*) on `int/real/char` shift & red. (pop 0, push *type*) on `array` shift and goto 5 on `record` shift and goto 10
5.	*type* ⟶ `array` • `const .. const of` *type*	on `const` shift and goto 6
6.	*type* ⟶ `array const` • `.. const of` *type*	on .. shift and goto 7
7.	*type* ⟶ `array const ..` • `const of` *type*	on `const` shift and goto 8
8.	*type* ⟶ `array const .. const` • `of` *type*	on `of` shift and goto 9
9.	*type* ⟶ `array const .. const of` • *type* ——————— *type* ⟶ • `int` | `real` | `char` *type* ⟶ • `array const .. const of` *type* *type* ⟶ • `record` *decl_list* `end`	on *type* shift and reduce (pop 5, push *type*) on `int/real/char` shift & red. (pop 0, push *type*) on `array` shift and goto 5 on `record` shift and goto 10
10.	*type* ⟶ `record` • *decl_list* `end` ——————— *decl_list* ⟶ • *decl_list decl* ; *decl_list* ⟶ • *decl* ; *decl* ⟶ • `id` : *type*	on *decl_list* shift and goto 11 on decl shift and goto 13 on `id` shift and goto 3
11.	*type* ⟶ `record` *decl_list* • `end` *decl_list* ⟶ *decl_list* • *decl* ; ——————— *decl* ⟶ • `id` : *type*	on `end` shift and reduce (pop 2, push *type*) on *decl* shift and goto 12 on `id` shift and goto 3
12.	*decl_list* ⟶ *decl_list decl* • ;	on ; shift and reduce (pop 2, push *decl_list*)
13.	*decl_list* ⟶ *decl* • ;	on ; shift and reduce (pop 1, push *decl_list*)

Using the CFSM, trace out a parse (as in Figure 2.30) for the following input program:

```
foo : record
        a : char;
        b : array 1 .. 2 of real;
    end;
```

Answer:

Parse stack	Input stream	Comment
0	`foo : record` ...	
0 `id` *3*	`: record a` ...	shift `id(foo)`
0 `id` *3* `:` *4*	`record a :` ...	shift `:`
0 `id` *3* `:` *4* `record` *10*	`a : char ;` ...	shift `record`
0 `id` *3* `:` *4* `record` *10* `id` *3*	`: char ;` ...	shift `id(a)`
0 `id` *3* `:` *4* `record` *10* `id` *3* `:` *4*	`char ; b :` ...	shift `:`
0 `id` *3* `:` *4* `record` *10* `id` *3* `:` *4*	*type* `; b :` ...	shift `char` & reduce by *type* ⟶ `char`
0 `id` *3* `:` *4* `record` *10*	*decl* `; b :` ...	shift *type* & reduce by *decl* ⟶ `id :` *type*
0 `id` *3* `:` *4* `record` *10* *decl* *13*	`; b :` ...	shift *decl*
0 `id` *3* `:` *4* `record` *10*	*decl_list* `b :` ...	shift `;` & reduce by *decl_list* ⟶ *decl* `;`
0 `id` *3* `:` *4* `record` *10* *decl_list* *11*	`b : array` ...	shift *decl_list*
0 `id` *3* `:` *4* `record` *10* *decl_list* *11* `id` *3*	`: array 1 .. 2` ...	shift `id(b)`
0 `id` *3* `:` *4* `record` *10* *decl_list* *11* `id` *3* `:` *4*	`array 1 .. 2` ...	shift `:`
0 `id` *3* `:` *4* `record` *10* *decl_list* *11* `id` *3* `:` *4* `array` *5*	`1 .. 2 of` ...	shift `array`
0 `id` *3* `:` *4* `record` *10* *decl_list* *11* `id` *3* `:` *4* `array` *5* `const` *6*	`.. 2 of` ...	shift `const(1)`
0 `id` *3* `:` *4* `record` *10* *decl_list* *11* `id` *3* `:` *4* `array` *5* `const` *6* `..` *7*	`2 of real ;` ...	shift `..`
0 `id` *3* `:` *4* `record` *10* *decl_list* *11* `id` *3* `:` *4* `array` *5* `const` *6* `..` *7* `const` *8*	`of real ;` ...	shift `const(2)`
0 `id` *3* `:` *4* `record` *10* *decl_list* *11* `id` *3* `:` *4* `array` *5* `const` *6* `..` *7* `const` *8* `of` *9*	`real ; end ;`	shift `of`
0 `id` *3* `:` *4* `record` *10* *decl_list* *11* `id` *3* `:` *4* `array` *5* `const` *6* `..` *7* `const` *8* `of` *9*	*type* `; end ;`	shift `real` & reduce by *type* ⟶ `real`
0 `id` *3* `:` *4* `record` *10* *decl_list* *11* `id` *3* `:` *4*	*type* `; end ;`	shift *type* & reduce by *type* ⟶ `array const .. const of` *type*
0 `id` *3* `:` *4* `record` *10* *decl_list* *11*	*decl* `; end ;`	shift *type* & reduce by *decl* ⟶ `id :` *type*
0 `id` *3* `:` *4* `record` *10* *decl_list* *11* *decl* *12*	`; end ;`	shift *decl*
0 `id` *3* `:` *4* `record` *10*	*decl_list* `end ;`	shift `;` & reduce by *decl_list* ⟶ *decl_list* *decl* `;`
0 `id` *3* `:` *4* `record` *10* *decl_list* *11*	`end ;`	shift *decl_list*
0 `id` *3* `:` *4*	*type* `;`	shift `end` & reduce by *type* ⟶ `record` *decl_list* `end`
0	*decl* `;`	shift *type* & reduce by *decl* ⟶ `id :` *type*
0 *decl* *13*	`;`	shift *decl*
0	*decl_list*	shift `;` & reduce by *decl_list* ⟶ *decl* `;`
[done]		

IN MORE DEPTH

2.31 Give an example of an erroneous program fragment in which consideration of semantic information (e.g., types) might help one make a good choice between two plausible "corrections" of the input.

Answer: In the following example

```
A := (B,C + D);
```

we might replace the comma with a plus sign:

```
A := (B+C + D);
```

Alternatively, we might replace the comma with a period:

```
A := (B.C + D);
```

This latter repair would make more sense if `B` is a record and `C` is one of its fields. Unfortunately, choosing between these repairs requires information from the semantic analysis phase of compilation, which may not be easily accessible (e.g., if that phase won't occur until a subsequent pass).

In a similar vein, most compilers decline to reinterpret characters that have been accepted by the scanner. If our example input had been

```
A := (3,2 + D);
```

then one possible repair would have been to turn the comma into a period and scan `3.2` as a floating-point constant, rather than as three separate tokens. Like the possibility of replacing `B,C` with `B.C`, this repair has the appealing property of reflecting a highly plausible error: the comma and period are right next to each other on a standard keyboard. The ease with which a programmer might type a comma instead of a period might cause us to favor the `B.C` repair above, even without semantic information. It is unlikely, however, that we should want to adopt the `3.2` repair, since we should have to get the scanner involved in order to reinterpret the already-accepted `3`. As in the previous paragraph, backing out might also require that we undo semantic actions.

2.32 Give an example of an erroneous program fragment in which the "best" correction would require one to "back up" the parser (i.e., to undo recent predictions/matches or shifts/reductions).

Answer: Continuing with the example of exercise C-2.31, we could insert an identifier in front of the left parenthesis, turning it into a function call:

```
A := F(B,C + D);
```

This repair might be the right one if the types of `B` and `C + D` match those of the parameters of a frequently used function `F`. Of course realizing this would again require semantic information. Moreover, it would require that we "back up" the input stream, since we have already accepted the left parenthesis and the identifier `B`. In a compiler that intermixes syntax analysis and semantic analysis, accepting these tokens may have caused us to modify the state not only of the parser, but of the semantic analyzer as well. In recognition of these complications, most compilers decline to back up the input. Given the absence of earlier errors, and the correctness of the parsing algorithm, we know that the tokens accepted so far are the prefix of *some* valid program. It is always possible to construct a syntactically valid program by modifying only those tokens that have not yet been accepted.

2.33 Extend your solution to exercise 2.21 to implement Wirth's syntax error recovery mechanism

- **(a)** with global FOLLOW sets, as in Example C-2.45.
- **(b)** with local FOLLOW sets, as in Example C-2.47.
- **(c)** with avoidance of "starter symbol" deletion, as in Example C-2.48.

2.34 Extend your solution to exercise 2.21 to implement exception-based syntax error recovery, as in Example C-2.49.

2.35 Prove that the grammars in Figure C-2.37 lie in the regions claimed.

Answer:

- LL(2) but not SLL:

  $S \longrightarrow$ `a` A `a` | `b` A `b a`

  $A \longrightarrow$ `b` | ϵ

  Recall that the difference between LL and SLL is local versus global follow sets. Realized as a PDA, an SLL parser has but two states, one of which is used only to accept; an LL parser has as many states as it needs to distinguish among local follow sets. Both parsing algorithms use the stack to hold the symbols expected in the future.

  The language generated by our grammar is finite: {`aa`, `aba`, `bba`, `bbba`}. Consider the case in which we have an A at top of stack and the remaining input is `b a`. What should we do? If we are in the middle of the first S production, we should clearly predict $A \longrightarrow$ `b`. If we are in the middle of the second S production, we should predict $A \longrightarrow \epsilon$. An LL(2) parser, which knows the context (i.e., whose PDA is in a different state depending on whether the previous match was an `a` or a `b`), and which can see the next two tokens of input, can clearly make this choice. An SLL parser, however, is stuck: because it does not remember its context, and because it can't peek below the A in the stack, it can't distinguish between being in the first production and being in the second. Since there are only two remaining tokens of input, no amount of look-ahead will help.

- SLL(k) but not LL($k-1$):

$S \longrightarrow \texttt{a}^{k-1}\ \texttt{b} \mid \texttt{a}^k$

With S at top of stack, k tokens of look-ahead allow us to distinguish between the two possible predictions; no smaller number suffices, and being LL doesn't help (there's only one context in the grammar!).

- LR(0) but not LL:

$S \longrightarrow A\ \texttt{b}$
$A \longrightarrow A\ \texttt{a} \mid \texttt{a}$

The CFSM for this grammar has five states:

0 $S \longrightarrow \bullet\ A\ \texttt{b}$

$A \longrightarrow \bullet\ A\ \texttt{a}$
$A \longrightarrow \bullet\ \texttt{a}$

1 $S \longrightarrow A \bullet\ \texttt{b}$
$A \longrightarrow A \bullet\ \texttt{a}$

2 $A \longrightarrow \texttt{a} \bullet$

3 $S \longrightarrow A\ \texttt{b} \bullet$

4 $A \longrightarrow A\ \texttt{a} \bullet$

In states 0 and 1 we shift, regardless of input. In states 2, 3, and 4 we reduce, regardless of input. The grammar is therefore LR(0). (The decision of which state to move to on a shift depends on the input; the decision of *whether* to shift does not.)

The language generated by the grammar is $\texttt{a}^i\texttt{b}$, where $i > 0$. Suppose the grammar is captured by an LL(k) parser, for some k. Consider inputs $X = \texttt{a}^{k-1}\texttt{b}$ and $Y = \texttt{a}^k\texttt{b}$. At the beginning of the input, with A at top of stack, the parser should predict $A \longrightarrow \texttt{a}$ for input X and $A \longrightarrow A\ \texttt{a}$ for input Y. In both cases, however, it will have exactly the same information: initial context, A at top of stack, and (at least) k a's of input. Since the transition function of an LL(k) parser is always deterministic, we have a contradiction.

- SLL(1) but not LALR:

$S \longrightarrow A\ \texttt{a} \mid B\ \texttt{b} \mid \texttt{c}\ C$
$C \longrightarrow A\ \texttt{b} \mid B\ \texttt{a}$
$A \longrightarrow D$
$B \longrightarrow D$
$D \longrightarrow \epsilon$

SLL(1) parsability is trivial: there are no prediction conflicts among non-epsilon productions, and the only epsilon production has no alternatives.

Now suppose there exists an LALR(k) parser, for some k. Consider the CFSM. The start state, 0, is

$S \longrightarrow \bullet\ A\ \texttt{a}$
$S \longrightarrow \bullet\ B\ \texttt{b}$
$S \longrightarrow \bullet\ \texttt{c}\ C$

$A \longrightarrow \bullet\ D$
$B \longrightarrow \bullet\ D$
$D \longrightarrow \bullet$

On either `a` or `b` we clearly reduce by $D \longrightarrow \epsilon$, and then shift the D. This takes us to the following state, 1:

$A \longrightarrow D\ \bullet$
$B \longrightarrow D\ \bullet$

There is another way to get to this state. Given a `c` in the start state, we will shift and go to the following state, 2:

$S \longrightarrow \texttt{c}\ \bullet\ C$

$C \longrightarrow \bullet\ A\ \texttt{b}$
$C \longrightarrow \bullet\ B\ \texttt{a}$
$A \longrightarrow \bullet\ D$
$B \longrightarrow \bullet\ D$
$D \longrightarrow \bullet$

In this state, given input `a` or `b`, we will clearly reduce by $D \longrightarrow \epsilon$ and then shift the D, taking us to state 1.

Now what do we do in state 1? Given input `a`, for example, we should reduce by $A \longrightarrow D$ if we came from state 0, but by $B \longrightarrow D$ if we came from state 2. Unfortunately we don't know where we came from. All we know is the state we're in and the input. Context-sensitive look-ahead doesn't help: there are ways of reaching this state in which either reduction is appropriate for input `a`. Arbitrary look-ahead doesn't help either: there's only one token of input remaining. We are left with a contradiction of the determinism of the transition function.

- SLL(k) and SLR(k) but not LR($k-1$):

$S \longrightarrow A\ \texttt{a}^{k-1}\ \texttt{b} \mid B\ \texttt{a}^{k-1}\ \texttt{c}$
$A \longrightarrow \epsilon$
$B \longrightarrow \epsilon$

SLL(k) is trivial. Consider then the CFSM. The start state is

$S \longrightarrow \bullet\ A\ \texttt{a}^{k-1}\texttt{b}$
$S \longrightarrow \bullet\ B\ \texttt{a}^{k-1}\texttt{c}$

$A \longrightarrow \bullet$
$B \longrightarrow \bullet$

With k tokens of look-ahead we can easily resolve the reduce-reduce conflict between $A \longrightarrow \epsilon$ and $B \longrightarrow \epsilon$. The rest of the machine is trivial, and the grammar is SLR(k). With fewer than k tokens of look-ahead, however, we cannot resolve the conflict in the state state, and because we are in the start state, there is no way that a refined notion of context (LALR or LR) can help.

- LALR(1) but not SLR:

$S \longrightarrow$ `b` A `b` | A `c` | `a b`

$A \longrightarrow$ `a`

The start state of the CFSM is

$S \longrightarrow \bullet$ `b` A `b`

$S \longrightarrow \bullet A$ `c`

$S \longrightarrow \bullet$ `a b`

$A \longrightarrow \bullet$ `a`

On an `a` we will shift and move to the following state:

$S \longrightarrow$ `a` $\bullet$ `b`

$A \longrightarrow$ `a` $\bullet$

At this point, given an input of `b`, do we shift or reduce? Because there is no other way to reach this state, and because a `b` cannot follow the particular A for which $A \longrightarrow \bullet$ `a` was added to the closure of the start state, we can be sure in an LALR parser that shifting is the correct action to take. In an SLR parser, however, we do not have access to such reasoning; all decisions are made on the basis of FOLLOW sets. And since `b` $\in$ FOLLOW(A), the SLR parser cannot resolve the shift-reduce conflict. Also, because the `b` in question can be the final token of the input in both cases, arbitrary look-ahead does not help.

- LR(1) but not LALR:

$S \longrightarrow$ `a` C `a` | `b` C `b` | `a` D `b` | `b` D `a`

$C \longrightarrow$ `c`

$D \longrightarrow$ `c`

If we compute sets of LR items, we begin in the following state, 0:

$S \longrightarrow \bullet$ `a` C `a`

$S \longrightarrow \bullet$ `b` C `b`

$S \longrightarrow \bullet$ `a` D `b`

$S \longrightarrow \bullet$ `b` D `a`

On an `a` we shift from state 0 to the following state, 1:

$S \longrightarrow$ `a` $\bullet$ `a` C `a`

$S \longrightarrow$ `a` $\bullet$ `a` D `b`

$C \longrightarrow \bullet$ `c`

$D \longrightarrow \bullet$ `c`

On a `b` we shift from state 0 to the following state, 2:

$S \longrightarrow \bullet$ `b` C `b`
$S \longrightarrow \bullet$ `b` D `a`

$C \longrightarrow \bullet$ `c`
$D \longrightarrow \bullet$ `c`

Then, on a `c`, from either state 1 or state 2 we shift to a state that looks like this:

$C \longrightarrow$ `c` $\bullet$
$D \longrightarrow$ `c` $\bullet$

Because the items are exactly the same in both cases, they will compose a single state in the CFSM of an LALR parser. This will force the look-aheads inherited from predecessor states to be merged, leaving a reduce-reduce conflict: both `a` and `b` are valid look-aheads for each item, in one predecessor or the other. An LR parser, by contrast, will have two different states, one for each predecessor. The look-aheads will not merge (they are reversed in the two different predecessors) and the conflict will not arise. As in previous examples, the finite nature of the language (only one more token of input remaining) means that extra look-ahead can't help.

- Unambiguous but not LR:

$S \longrightarrow$ `a` S `a` $\mid \epsilon$

This grammar generates all nonempty strings consisting of an even number of `a`'s. Suppose it had an LR(k) parser, for some k, and consider the inputs $X = \mathtt{a}^{2k}$ and $Y = \mathtt{a}^{2k+2}$. After having parsed k `a`'s, we must be in a state containing the following LR items:

$S \longrightarrow$ `a` $\bullet$ S `a`

$S \longrightarrow \bullet$ `a` S `a`
$S \longrightarrow \bullet$

Our paths cannot have been different on the two different inputs, because at every point we will have had the exact same information available. Seeing another `a` on the input, do we shift or reduce? On input X we should reduce by $S \longrightarrow \epsilon$; on input Y we should shift. In both cases, however, we will still have identical information: same current state; same k upcoming tokens. Since the transition function is deterministic, we have a contradiction.

2.36 (Difficult) Prove that the languages in Figure C-2.39 lie in the regions claimed.

Answer:

Nondeterministic language:

$L_1 = \{\mathtt{a}^n\mathtt{b}^n\mathtt{c} : n \geq 1\} \cup \{\mathtt{a}^n\mathtt{b}^{2n}\mathtt{d} : n \geq 1\}$

Our proof is easier if we first show that the language $X3 = \mathtt{a}^i\mathtt{b}^i\mathtt{c}^i$ is not context free. Suppose the contrary. Let G be an arbitrary grammar for $X3$. Let x be the number of nonterminals in G. Let y be the maximum number of symbols on any right-hand side in G. Let $n = y^x + 1$. Consider a parse tree T_1 for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$. Call an instance of a nonterminal in T_1 *productive* if its yield is longer than those of any of its children. Clearly any subtree of

T_1 with n leaves must have a leaf f with $h > x$ productive nonterminals above it. By the pigeonhole principle some nonterminal X must have at least two productive occurrences on the path p from the root of T_1 to f. Among all nonterminals that repeat on p, let X be the one whose second-to-bottom occurrence is closest to the leaves. (Below the upper of the two bottom occurrences of X, no nonterminal is repeated on p.) The yield of the lower X must be less than n terminals in length, so it must be missing either `a`'s or `c`'s. If we replace the subtree rooted at the lower X with (a copy of) the subtree rooted at the upper X we obtain a valid parse tree T_2 whose yield contains a different number of nonterminals, total, but the same number of either `a`'s or `c`'s. This implies that the yield of T_2 is not in $\mathtt{a}^i\mathtt{b}^i\mathtt{c}^i$, a contradiction.

Now suppose that L_1 can be parsed deterministically. Then there is a deterministic PDA M_1 that accepts it. Use M_1 to construct a nondeterministic PDA as follows: Create a copy of M_1; call it M_2. In any state of M_1 that has a transition on `c` to a final state, create an epsilon transition to the corresponding state of M_2. Relabel all transitions on `b` in M_2 to be transitions on `c` instead. In any state of M_2 that has a transition on `d` to a final state, create an epsilon transition to a new state s. Change all final states of M_1 and M_2 to be non-final. Make s final. Make the start state of M_1 the start state of the new combined machine. This new machine clearly accepts $X3$, a contradiction.

Inherently ambiguous language:

$L_2 = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k : i = j \text{ or } j = k;\ i, j, k \geq 1\}$

This proof is very difficult. The intuition is that once n is sufficiently large to preclude special treatment in the grammar, there have to be two different ways to build a parse tree for $\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n$—one that ensures an equal number of `a`'s and `b`'s, another that ensures an equal number of `b`'s and `c`'s. A rigorous treatment was published by Herman Maurer in the April 1969 issue of the *Journal of the ACM*. It's five pages long.

Language with LL(k) grammar but no LL($k-1$) grammar:

$L_3 = \{\mathtt{a}^n(\ \mathtt{b}\ \mid\ \mathtt{c}\ \mid\ \mathtt{b}^k\mathtt{d}\ )^n : n \geq 1\}$

This language clearly has an LL(k) grammar:

$S \longrightarrow \mathtt{a}\ B\ L$

$B \longrightarrow \mathtt{a}\ B\ L \mid \epsilon$

$L \longrightarrow \mathtt{b} \mid \mathtt{c} \mid \mathtt{b}^k\ \mathtt{d}$

The k tokens of lookahead suffice to choose among the L productions. The other half of the proof—that L_3 cannot have an LL($k-1$) grammar—is very difficult. The general outline appears below; for details see Section 8.1.4 of Aho and Ullman's classic two-volume reference [AU72].

Lemma 1 If a language L has an LL(k) grammar without epsilon productions, then it has an LL(k) grammar in which every production has the form $A \longrightarrow \mathtt{a}\ \alpha$, where `a` is a terminal and α is a string of zero or more nonterminals. Such a grammar is said to be in *Griebach Normal Form* (GNF).

Lemma 2 Every LL(k) grammar for L_3 must contain at least one epsilon production. Otherwise it would have an LL(k) grammar in GNF. One can show that the existence of such a grammar would permit the derivation of a string containing a `d` preceded by only $k-1$ `b`'s, a contradiction.

Lemma 3 For every LL($k-1$) grammar there is an LL(k) grammar without epsilon productions that generates the same language.

Main theorem If L_3 had an LL($k-1$) grammar then it would have an LL(k) grammar without epsilon productions, a contradiction of Lemma 2.

Language with LR(0) grammar but no LL grammar:

$L_4 = \{\mathtt{a}^n\mathtt{b}^n : n \geq 1\} \cup \{\mathtt{a}^n\mathtt{c}^n : n \geq 1\}$

The LR(0) grammar is trivial:

$S \longrightarrow B \mid C$

$B \longrightarrow \mathtt{a\ b} \mid \mathtt{a}\ B\ \mathtt{b}$

$C \longrightarrow \mathtt{a\ c} \mid \mathtt{a}\ B\ \mathtt{c}$

Now suppose that L_4 has an LL(k) grammar for some k. From the discussion of L_3, L_4 has an LL(k) grammar G in GNF. Using this grammar, the derivation of $\mathtt{a}^{i+k}\ \mathtt{b}^{i+k}$ is of the form $S \Longrightarrow^* \mathtt{a}^i\ \alpha_i \Longrightarrow^* \mathtt{a}^{i+k}\ \mathtt{b}^{i+k}$. By choosing i large enough, we can ensure that α_i is of the form $\beta B\gamma$, where $|\beta| \geq k$ and $|\gamma| \geq k$. And since G has no epsilon productions, we know that $B \Longrightarrow^* \mathtt{b}^l$, where $l \geq 1$.

Now consider the derivation of $\mathtt{a}^{i+k}\ \mathtt{c}^{i+k}$. By similar reasoning we must have $S \Longrightarrow^* \mathtt{a}^i\ \delta C\eta$, where $|\delta| \geq k$ and $|\eta| \geq k$, and where $C \Longrightarrow^* \mathtt{c}^m$ for some $m \geq 1$.

But given the limit of k tokens of lookahead, our parser must be in a unique state after reading $\mathtt{a}^i$, so $\beta B\gamma = \delta C\eta = \alpha_i$. In other words, the yield of α_i contains at least one `b` *and* at least one `c`, a string that is not a suffix of any string in the language. Our assumption that L_4 has an LL(k) grammar must therefore be incorrect.

2.37 Prove that regular expressions and *left-linear grammars* are equally powerful. A left-linear grammar is a context-free grammar in which every right-hand side contains at most one nonterminal, and then only at the left-most end.

Answer: Since we know that regular expressions and finite automata are equally powerful, it suffices to show that left linear grammars and finite automata are equally powerful. This in turn can be shown by giving constructions to turn a DFA into a left-linear grammar and vice versa.

Suppose we are given a DFA. Assign a nonterminal name to each state. Then, for each transition from, say, state X to state Y on input symbol `a`, create a production $Y \longrightarrow X\ \mathtt{a}$. Create one more production $I \longrightarrow \epsilon$, where I is the start state. Finally, for each final state F ($F \neq S$), create a production $S \longrightarrow F$. If the start state is final, create a production $S \longrightarrow \epsilon$. The resulting grammar, with start symbol S, is clearly left-linear, and equivalent to the DFA.

Now suppose we are given a left-linear grammar. We first modify it so that every production contains at most one input symbol. Specifically, we replace any production $A \longrightarrow B\ \mathtt{a}_1 \ldots \mathtt{a}_{n-1}\ \mathtt{a}_n$ with

$A \longrightarrow C_{n-1}\ \mathtt{a}_n$

$C_{n-1} \longrightarrow C_{n-2}\ \mathtt{a}_{n-1}$

$\ldots$

$C_2 \longrightarrow C_1\ \mathtt{a}_2$

$C_1 \longrightarrow B\ \mathtt{a}_1$

where the C's are new nonterminals. (If B is missing in the original production we leave it out of the final new production.) We now create a finite automaton with one state for every nonterminal in the modified grammar, and one additional start state. The state labeled with the start symbol of the grammar is the (only) final state. For every production of the form $A \longrightarrow$ `a` we create an arc labeled `a` from the start state to state A. For every production of the form $A \longrightarrow B$ `a` we create an arc labeled `a` from state B to state A. For every production of the form $A \longrightarrow B$ we create an arc labeled ϵ from state B to state A. The resulting NFA is clearly equivalent to the given left-linear grammar. It can of course be converted to a DFA if desired.

3 Names, Scopes, and Bindings

3.11 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

3.1 Indicate the binding time (when the language is designed, when the program is linked, when the program begins execution, etc.) for each of the following decisions in your favorite programming language and implementation. Explain any answers you think are open to interpretation.

- The number of built-in functions (math, type queries, etc.)
- The variable declaration that corresponds to a particular variable reference (use)
- The maximum length allowed for a constant (literal) character string
- The referencing environment for a subroutine that is passed as a parameter
- The address of a particular library routine
- The total amount of space occupied by program code and data

Answer: Here are answers for C:

The number of built-in functions is originally bound at language design time, though it may be increased by certain implementations. C has just a few functions that are truly built-in, notably `sizeof`. A large number of additional functions are defined by the standard library. Several of these, including `printf`, `malloc`, `assert`, and the various `stdarg` routines, are often special-cased by the compiler in order to generate faster or safer code.

The variable declaration that corresponds to a particular variable reference (use) is bound at compile time: C uses static scope.

The maximum length of a character string (if there is a limit) is bound at language implementation time.

Because C does not have nested subroutines, the referencing environment for a subroutine that is passed as a parameter is always the same as the environment in effect when the subroutine was declared.

The address of a particular library function is bound by the linker in most systems, though it may not be known until load time or even run time in systems that perform dynamic linking (Section C-15.7). Note that we're speaking here of virtual addresses; physical addresses are invisible to the running program, and are often changed by the operating system during execution).

The total amount of space occupied by program code and data is bound at run time: the amount of stack and heap space needed will often depend on the input.

3.2 In Fortran 77, local variables were typically allocated statically. In Algol and its descendants (e.g., Ada and C), they are typically allocated in the stack. In Lisp they are typically allocated at least partially in the heap. What accounts for these differences? Give an example of a program in Ada or C that would not work correctly if local variables were allocated statically. Give an example of a program in Scheme or Common Lisp that would not work correctly if local variables were allocated on the stack.

Answer: Fortran 77 lacks recursion, so there can never be more than one live instance of the local variables of a given subroutine. Algol and its descendants require a stack to accommodate multiple copies. The following would not work correctly with statically allocated local variables:

```
function sum(f, low, high) if low = high return f(low) else return f(low) + sum(f, low + 1, high)
```

Function `sum` needs to remember the value of `low` during the recursive call. (As we shall see in Section 6.6.1, it is possible to write a *tail recursive* version of `sum` that does not need to remember anything during the recursive call, but only if we are willing to change the calling signature of the function or to exploit the associativity of addition.)

Algol and its descendants (for the most part) have limited extent for local variables, meaning that the values of those variables are lost when control leaves the scope in which they were declared. Lisp and its descendants require allocation in the heap to accommodate unlimited extent. The following would not work correctly with stack-allocated variables:

```
function add_n(n) return { function(k) return n + k }
```

The intent here is to return a function which, when called, will add to its argument `k` the value `n` originally passed to `add_n`. That value (`n`) must remain accessible as long as the function returned by `add_n` remains accessible.

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.

Answer: There are many possible answers. Here are a few:

Just-in-time compilation allows a system to minimize code size, obtain the most recent implementations of standard abstractions, and avoid the overhead of compiling functions that aren't used. Dynamic linking (Section C-15.7) has similar advantages over static linking. (Just-in-time compilation also allows us to ship machine-independent code around the Internet, but in that case we don't have enough information to bind early.)

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.

```
1.  procedure main()
2.      a : integer := 1
3.      b : integer := 2

4.      procedure middle()
5.          b : integer := a

6.          procedure inner()
7.              print a, b

8.          a : integer := 3

9.          -- body of middle
10.         inner()
11.         print a, b

12.     -- body of main
13.     middle()
14.     print a, b
```

Suppose this was code for a language with the declaration-order rules of C (but with nested subroutines)—that is, names must be declared before use, and the scope of a name extends from its declaration through the end of the block. At each print statement, indicate which declarations of a and b are in the referencing environment. What does the program print (or will the compiler identify static semantic errors)? Repeat the exercise for the declaration-order rules of C# (names must be declared before use, but the scope of a name is the entire block in which

it is declared) and of Modula-3 (names can be declared in any order, and their scope is the entire block in which they are declared).

Answer: With C rules, line 7 refers to the a and b declared on lines 2 and 5, respectively; line 11 refers to the a and b declared on lines 8 and 5, respectively; and line 14 refers to the a and b declared on lines 2 and 3, respectively. The program prints 1 1 3 1 1 2. With C# rules, the compiler should produce use-before-define errors for a at lines 5 and 7. With Modula-3 rules, line 7 refers to the a and b declared on lines 8 and 5, respectively; line 11 refers to the a and b declared on lines 8 and 5, respectively; and line 14 refers to the a and b declared on lines 2 and 3, respectively. The program prints 3 3 3 3 1 2.

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

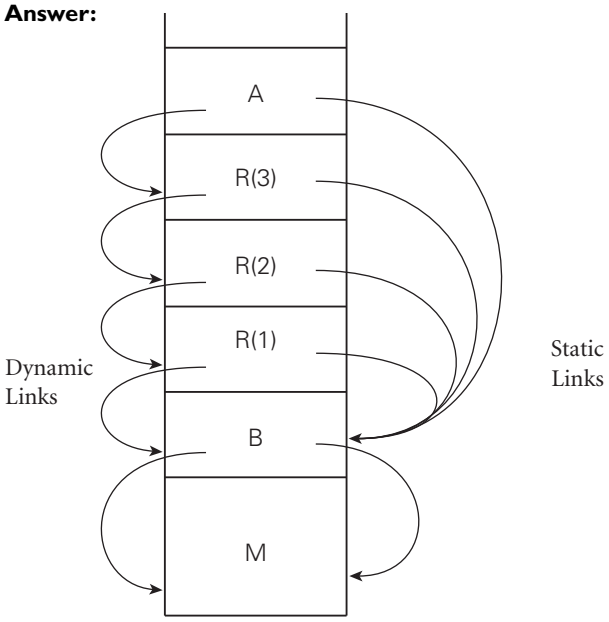
```
procedure main()
    g : integer

    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 3.

(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.)

3.7 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 3.16.

(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()) {
    L = insert(next_widget(), L);
}
L = reverse(L);
```

Sadly, after running for a while, Brad's program always runs out of memory and crashes. Explain what's going wrong.

Answer: The `reverse_list` routine produces a new list, composed of new list nodes. When Brad assigns the return value back into `L` he loses track of the old list nodes, and never reclaims them. In other words, his program has a memory leak. After some number of iterations of his main loop, Brad has exhausted the heap and his program can't continue.

```
typedef struct list_node {
    void* data;
    struct list_node* next;
} list_node;

list_node* insert(void* d, list_node* L) {
    list_node* t = (list_node*) malloc(sizeof(list_node));
    t->data = d;
    t->next = L;
    return t;
}

list_node* reverse(list_node* L) {
    list_node* rtn = 0;
    while (L) {
        rtn = insert(L->data, rtn);
        L = L->next;
    }
    return rtn;
}

void delete_list(list_node* L) {
    while (L) {
        list_node* t = L;
        L = L->next;
        free(t->data);
        free(t);
    }
}
```

Figure 3.16 List management routines for Exercise 3.7.

(b) After Janet patiently explains the problem to him, Brad gives it another try:

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

This seems to solve the insufficient memory problem, but where the program used to produce correct results (before running out of memory), now its output is strangely corrupted, and Brad goes back to Janet for advice. What will she tell him this time?

Answer: While the call to `delete_list` successfully reclaims the old list nodes, it also reclaims the widgets. The new, reversed list thus contains dangling references. These refer

to locations in the heap that may be used for newly allocated data, which may be corrupted by uses of the elements in the reversed list. Brad seems to have been lucky in that he isn't corrupting the heap itself (maybe his widgets are the same size as list nodes), but without Janet's help he may have a lot of trouble figuring out why widgets are changing value "spontaneously."

3.8 Rewrite Figures 3.6 and 3.7 in C. You will need to use separate compilation for name hiding.

Answer:

3.9 Consider the following fragment of code in C:

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

(a) Assume that each integer variable occupies four bytes. How much total space is required for the variables in this code?

Answer: Variables `a`, `b`, and `c` are live throughout the execution of the outer block. Variables `d`, `e`, and `f` are needed only in the first nested block, and can overlap the space devoted to `g`, `h`, and `i`. A total of $4 \times 6 = 24$ bytes is required.

(b) 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.

Answer: When compiling a subroutine, the compiler can construct a tree in which each node represents a block, and is a child of the node that represents the surrounding block. Variables declared in the outermost block are assigned locations at the beginning of the subroutine's space. Variables in a nested block are assigned locations immediately following the variables of the surrounding (parent) block. Variables of siblings overlap.

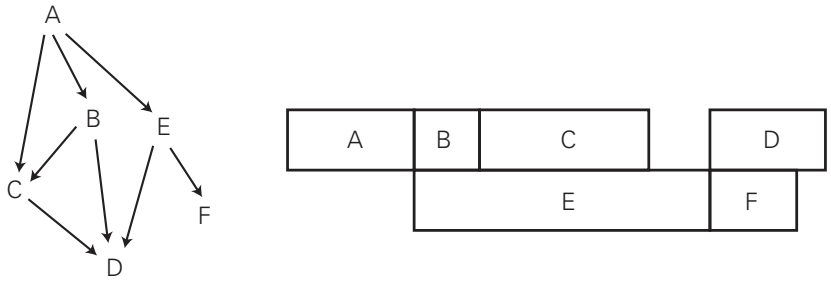
3.10 Consider the design of a Fortran 77 compiler that uses static allocation for the local variables of subroutines. Expanding on the solution to the previous question, describe an algorithm to minimize the total space required for these variables. You

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.

Answer: Since Fortran 77 lacks recursion, the call graph for a Fortran 77 program is guaranteed to be acyclic. If two subroutines never appear on the same path from `main` in this graph, then their local variables will never be needed at the same time, and may in fact share space.

If our compiler allocates local variables statically, we can choose a minimum-size allocation by performing a topological sort of the call graph. The "frame" of any subroutine that can be called *only* from the main program is placed at address 0. After this, the frame of any subroutine that can be called only by subroutines whose frames have already been assigned locations is placed at the first address that is beyond the frames of all of its potential callers.

As an example, consider the acyclic call graph below. It shows that subroutine `E` is never active at the same time as `B` or `C`. When allocating space we can therefore overlap `E`'s storage with that of `B` and `C`. Similarly, the space for `D` and `F` can overlap, though both must be disjoint from `A` and `E`. (The vertical offsetting of boxes is for clarity of presentation only.)



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)`.

3.13 Consider the following program in Scheme:

```
(define A
  (lambda()
    (let* ((x 2)
           (C (lambda (P)
                (let ((x 4))
                  (P))))
           (D (lambda ()
                x))
           (B (lambda ()
                (let ((x 3))
                  (C D)))))
      (B))))
```

What does this program print? What would it print if Scheme used dynamic scoping and shallow binding? Dynamic scoping and deep binding? Explain your answers.

Answer: In standard Scheme the program prints a 2. Scheme uses static scope and deep binding, so `D` sees the `x` in `A`.

With dynamic scoping and shallow binding, the program would print a 4. `D` is eventually called from `C`, so the most recently declared `x` is the one in `C`.

With dynamic scoping and deep binding, the program would print a 3. `D` is bound to a referencing environment in `B`, when it is passed to `C`. At that point the most recently declared `x` is the one in `B`.

3.14 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?

Answer: With static scoping it prints `1 1 2 2`. With dynamic scoping it prints `1 1 2 1`. The difference lies in whether set_x sees the global x or the x declared in second when it is called from second.

3.15 The principal argument in *favor* of dynamic scoping is that it facilitates the customization of subroutines. Suppose, for example, that we have a library routine print_integer that is capable of printing its argument in any of several bases (decimal, binary, hexadecimal, etc.). Suppose further that we want the routine to use decimal notation most of the time, and to use other bases only in a few special cases: we do not want to have to specify a base explicitly on each individual call. We can achieve this result with dynamic scoping by having print_integer obtain its base from a nonlocal variable print_base. We can establish the default behavior by declaring a variable print_base and setting its value to 10 in a scope encountered early in execution. Then, any time we want to change the base temporarily, we can write:

```
begin       -- nested block
    print_base : integer := 16       -- use hexadecimal
    print_integer(n)
```

The problem with this argument is that there are usually other ways to achieve the same effect, without dynamic scoping. Describe at least two for the print_integer example.

Answer:

One option would be to have print_integer use decimal notation in all cases, and create another routine, print_integer_with_base, that takes a second argument. In a language like Ada or C++, one could make the base an optional (default) parameter of a single print_integer routine, or use overloading to give the same name to both routines. (Default parameters are discussed in Section 9.3.3; overloading is discussed in Section 3.5.2.)

A second option would be to create a static variable, either global or encapsulated with print_integer inside an appropriate module, that controls the base. To change the print base temporarily, we coould then write

```
begin       -- nested block
    print_base_save : integer := print_base
    print_base := 16       -- use hexadecimal
    print_integer(n)
    print_base := print_base_save
```

The possibility that we may forget to restore the original value, of course, is a potential source of bugs. With dynamic scoping the value is restored automatically.

3.16 As noted in Section 3.6.3, C# has unusually sophisticated support for first-class subroutines. Among other things, it allows *delegates* to be instantiated from anonymous nested methods, and gives local variables and parameters unlimited extent when they may be needed by such a delegate. Consider the implications of these features in the following C# program.

```
using System;
public delegate int UnaryOp(int n);
   // type declaration: UnaryOp is a function from ints to ints

public class Foo {
    static int a = 2;
    static UnaryOp b(int c) {
        int d = a + c;
        Console.WriteLine(d);
        return delegate(int n) { return c + n; };
    }
    public static void Main(string[] args) {
        Console.WriteLine(b(3)(4));
    }
}
```

What does this program print? Which of `a`, `b`, `c`, and `d`, if any, is likely to be statically allocated? Which could be allocated on the stack? Which would need to be allocated in the heap? Explain.

Answer: The program prints

```
5
7
```

Objects `a` and `b` are globally defined, and could be statically allocated. (The code for the anonymous function returned by `b` could likewise be statically allocated.) Local variable `d` could be allocated on the stack. While parameters can often be allocated on the stack, `c` has unlimited extent (it is needed by the anonymous function returned by `b`), so it has to go in the heap.

3.17 If you are familiar with structured exception handling, as provided in Ada, 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 9.4.)

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.

3.18 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
```

3.19 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.

3.20 Consider mathematical operations in a language like C++, which supports both overloading and coercion. In many cases, it may make sense to provide multiple, overloaded versions of a function, one for each numeric type or combination of types. In other cases, we might use a single version—probably defined for double-precision floating point arguments—and rely on coercion to allow that function to be used for other numeric types (e.g., integers). Give an example in which overloading is clearly the preferable approach. Give another in which coercion is almost certainly better.

Answer: Consider a function to return the lesser of two floating-point arguments:

```
double min(double x, double y) { ...
```

If this function is called in a context that expects an integer (e.g., `i = min(j, k)`), the compiler will coerce the integer arguments (`j` and `k`) to floating-point numbers, call `min`, and then coerce the result back to an integer (via truncation). While the result may be numerically correct (assuming `double` variables have at least as many significant bits as `int`s), it won't be very fast. An overloaded interger-specific version of `min` would use integer arithmetic for the comparison (which may be cheaper in and of itself), and would avoid three conversion operations.

On the flip side, consider the following function, which 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));
}
```

Here `s` has a 50/50 chance of being nonintegral even if `a`, `b`, and `c` are all integers, and the result is almost certainly nonintegral, so integer-to-floating-point conversions are almost certainly going to be needed in any case. Moreover the possibility that any subset of `{a, b, c}` might be integral means that eight different functions would be needed to cover all combinations, which seems excessive.

3.21 In a language that supports operator overloading, build support for rational numbers. Each number should be represented internally as a (numerator, denominator) pair in simplest form, with a positive denominator. Your code should support unary negation and the four standard arithmetic operators. For extra credit, create a conversion routine that accepts two floating-point parameters—a value and a error bound—and returns the simplest (smallest denominator) rational number within the given error bound of the given value.

Answer: Here's a solution in C++:

```
// Simple class for rational numbers in C++.
// Fractions are always kept in reduced form.
// Denominator is always positive.
// Does not deal with divide-by-zero or overflow.

#include <cmath>
    using std::abs;
    using std::round;
#include <iostream>
    using std::ostream;
    using std::cout;

class rational {
    int num;
    int den;
    int gcd(int i, int j) {
        i = abs(i);
        j = abs(j);
        while (i != j) {
            if (i > j) i = i - j;
            else j = j - i;
        }
        return i;
    }
    void normalize() {
        if (den < 0) {
            num = -num;
            den = -den;
        }
        int g = gcd(num, den);
        num /= g;
        den /= g;
    }
public:
    rational(int n, int d) {
        num = n;
        den = d;
        normalize();
    }
    // initialize to simplest fraction within e of r
    rational(double r, double e) {
        num = (int) r;
        den = 1;
        while (abs(r - ((double)num)/((double)den)) > e) {
            ++den;
            num = round(den*r);
        }
    }
```

```
    rational operator-() {
        num = -num;
        return *this;
    }
    rational operator+(const rational o) {
        int p = den * o.den;
        num = num * o.den + den * o.num;
        den = p;
        normalize();
        return *this;
    }
    rational operator-(const rational o) {
        int p = den * o.den;
        num = num * o.den - den * o.num;
        den = p;
        normalize();
        return *this;
    }
    rational operator*(const rational o) {
        num *= o.num;
        den *= o.den;
        normalize();
        return *this;
    }
    rational operator/(const rational o) {
        num *= o.den;
        den *= o.num;
        normalize();
        return *this;
    }
};
```

3.22 In an imperative language with lambda expressions (e.g., C#, Ruby, C++, or Java), write the following higher-level functions. (A higher-level function, as we shall see in Chapter 11, takes other functions as argument and/or returns a function as a result.)

- `compose(g, f)`—returns a function `h` such that `h(x) == g(f(x))`.
- `map(f, L)`—given a function `f` and a list `L` returns a list `M` such that the *i*th element of `M` is `f(`*e*`)`, where *e* is the *i*th element of `L`.
- `filter(L, P)`—given a list `L` and a predicate (Boolean-returning function) `P`, returns a list containing all and only those elements of `L` for which `P` is true.

Ideally, your code should work for any argument or list element type.

Answer: Here are solutions in C++:

```
#include <list>
    using std::list;
#include <functional>
    using std::function;

template<typename T>
function<T(T)> compose(function<T(T)>g, function<T(T)>f) {
    return [=](T e){ return f(g(e)); };
}

template<typename T>
list<T> map(list<T> L, function<T(T)> f) {
    list<T> R;
    for (auto & e : L) {
        R.push_back(f(e));
    }
    return R;
}

template<typename T>
list<T> filter(list<T> L, function<bool(T)> P) {
    list<T> R;
    for (auto & e : L) {
        if (P(e)) R.push_back(e);
    }
    return R;
}
```

3.23 Can you write a macro in standard C that "returns" the greatest common divisor of a pair of arguments, without calling a subroutine? Why or why not?

Answer: No. The macro must either call a subroutine or execute a loop. It must also take the form of an expression in order to return a value. C provides no way for an expression to contain a loop. Many C implementations, however (including `gcc`) extend the language to include "statement expressions" that eliminate this restriction. The following macro works with `gcc`:

```
#define GCD(a, b)                       \
({  while ((a) != (b)) {                \
        if ((a) > (b)) (a) = (a) - (b); \
        else (b) = (b) - (a);           \
    }                                   \
    (a);                                \
})
```

IN MORE DEPTH

3.24 Assuming a LeBlanc-Cook style symbol table, explain how the compiler finds the symbol table information (e.g., the type) of a complicated reference such as `my_firm->revenues[1999]`.

Answer: The compiler begins by looking up `my_firm` in the usual way: traverse the scope stack from top to bottom; for each scope, look for ⟨scope, `my_firm`⟩ in the hash table; when found, retrieve the indicated symbol table entry, e_1. Entry e_1 will include a reference to the entry e_2 for the type of `my_firm`. Since this is a pointer type, e_2 will include a reference to the entry e_3 for the pointed-at type. Since this type is a record, e_3 will include the id r of the record's scope. The compiler will then look for ⟨r, `revenues`⟩ in the hash table. Assuming the name is valid, the search will hit, and return a reference to the symbol table entry e_4 of `revenues`. Entry e_4 will in turn refer to the entry e_5 of its type, which will be an array. Finally, entry e_5 will include references to the types of the subscript (integer) and element types. Combined, entries e_1–e_5 contain all the information required to type-check uses of `my_firm->revenues[1999]` and to generate code for it.

3.25 Show the contents of a LeBlanc-Cook style symbol table that captures the referencing environment of

(a) function `F1` in Figure 3.4.

Answer: The main hash table contains the following entries (order depends on hash function).

	name	category	scope	type	other
1	`real`	type	0	–	
2	`integer`	type	0	–	
3	`T1`	type	1	–	
4	`T2`	type	1	–	
5	`T3`	type	1	–	
6	`T4`	type	1	–	
7	`T5`	type	1	–	
8	`T6`	type	1	–	
9	`P1`	proc	1	–	params: 10
10	`A1`	param	2	3	
11	`X`	var	2	1	
12	`X`	var	6	2	
13	`P2`	proc	2	–	params: 14
14	`A2`	param	3	4	
15	`P3`	proc	3	–	params: 16
16	`A3`	param	4	5	
17	`P4`	proc	2	–	params: 18
18	`A4`	param	5	6	
19	`F1`	func	5	8	params: 20
20	`A5`	param	6	7	

The scope stack when compiling the body of `F1` looks as follows.

scope	closed?	
6	N	`F1`
5	N	`P4`
2	N	`P1`
1	N	globals
0	N	built-in

(b) procedure `set_seed` in Figure 3.7.

Answer:

3.26 Consider the visibility of class members (fields and methods) in an object-oriented language, as discussed near the end of Section C-3.4.1. Describe a mechanism that could be used to check visibility after first locating the member in a more traditional symbol table. (You may want to look ahead to Section 10.2.2.)

Answer:

3.27 Show a trace of the contents of the referencing environment A-list during execution of the program in

(a) Figure 3.9. Assume that a positive value is read at line 8.

Answer:

initially
    `integer`
elaborate global declarations
    `second`, `first`, $\texttt{a}_{\text{global}}$, `integer`
call `second` (line 8)
    $\texttt{a}_{\text{second}}$, `second`, `first`, $\texttt{a}_{\text{global}}$, `integer`
call `first` (line 6)
    $\texttt{a}_{\text{second}}$, `second`, `first`, $\texttt{a}_{\text{global}}$, `integer`
(so now when we set `a` to 1 at line 3 we're using the newer `a`)
return from `first`
    $\texttt{a}_{\text{second}}$, `second`, `first`, $\texttt{a}_{\text{global}}$, `integer`
return from `second`
    `second`, `first`, $\texttt{a}_{\text{global}}$, `integer`
(and now when we write `a` at line 12 we're using $\texttt{a}_{\text{global}}$, which is still 2)

(b) Exercise 3.14.

Answer: (For brevity, lines in which the environment does not change are not shown.)

initially
    `integer`
elaborate global declarations
    `second`, `first`, `print_x`, `set_x`, $\texttt{x}_{\text{global}}$, `integer` call `set_x(0)`
return from `set_x`
call `first`
call `set_x(1)`
return from `set_x`

call `print_x` (prints global `x`)
return from `print_x`
return from `first`
call `print_x` (prints global `x`)
return from `print_x`
call `second`
x_{second}, `second`, `first`, `print_x`, `set_x`, x_{global}, `integer`
call `set_x(2)`
return `from set_x`
call `print_x` (prints `second`'s `x`)
return from `print_x`
return from `second`
`second`, `first`, `print_x`, `set_x`, x_{global}, `integer`
call `print_x` (prints global `x`)
return from `print_x`

3.28 Repeat the previous exercise for a central reference table.

Answer:

(a) initially

integer	`integer`
first	–
second	–
a	–

elaborate global declarations

integer	`integer`
first	`first`
second	`second`
a	a_{global}

call `second` (line 8)

integer	`integer`
first	`first`
second	`second`
a	a_{second}, a_{global}

call `first` (line 6)

integer	`integer`
first	`first`
second	`second`
a	a_{second}, a_{global}

return from `first`

integer	`integer`
first	`first`
second	`second`
a	a_{second}, a_{global}

return from `second`

integer	`integer`
first	`first`
second	`second`
a	a_{global}

(b) (Table is shown only at points where it changes.)

initially

integer	`integer`
set_x	–
print_x	–
first	–
second	–
x	–

elaborate global declarations

integer	`integer`
set_x	`set_x`
print_x	`print_x`
first	`first`
second	`second`
x	x_{global}

call `set_x(0)`
return from `set_x`
call `first`
call `set_x(1)`
return from `set_x`
call `print_x` (prints global `x`)
return from `print_x`
return from `first`
call `print_x` (prints global `x`)
return from `print_x`
call `second`

integer	`integer`
set_x	`set_x`
print_x	`print_x`
first	`first`
second	`second`
x	x_{second}, x_{global}

call `set_x(2)`
return `from set_x`
call `print_x` (prints `second`'s `x`)
return from `print_x`
return from `second`

integer	`integer`
set_x	`set_x`
print_x	`print_x`
first	`first`
second	`second`
x	x_{global}

call `print_x` (prints global `x`)
return from `print_x`

3.29 Consider the following tiny program in C:

```
void hello() {
    printf("Hello, world\n");
}

int main() {
    hello();
}
```

(a) Split the program into two separately compiled files, `tiny.c` and `hello.c`. Be sure to create a header file `hello.h` and include it correctly in `tiny.c`.

Answer: File `tiny.c`:

```
#include "hello.h"

int main() {
    hello();
}
```

File `hello.h`:

```
extern void hello();
```

File `hello.c`:

```
void hello() {
    printf("Hello, world\n");
}
```

(b) Reconsider the program as C++ code. Put the `hello` function in a separate namespace, and include an appropriate `using` declaration in `tiny.c`.

Answer: File `tiny.cc`:

```
#include "hello.h"
using NS_hello::hello;

int main() {
    hello();
}
```

File `hello.h`:

```
namespace NS_hello {
    extern void hello();
}
```

File `hello.cc`:

```
#include <stdio.h>

namespace NS_hello {
    void hello() {
        printf("Hello, world\n");
    }
}
```

(c) Rewrite the program in Java, with `main` and `hello` in separate packages.

Answer: File `Tiny.java`:

```
import Hello.Greeting;

class Tiny {
    public static void main(String[] args) {
        Greeting.hello();
    }
}
```

File `Greeting.java`, in subdirectory `Hello`:

```
package Hello;

public class Greeting {
    public static void hello() {
        System.out.println("Hello, world");
    }
}
```

3.30 Consider the following file from some larger C program:

```
int a;
extern int b;
static int c;

void foo() {
    int a;
    static int b;
    extern int c;
    extern int d;
}

static int b;
extern int c;
```

For each variable declaration, indicate whether the variable has external linkage, internal (file-level) linkage, or no linkage (i.e., is local).

Answer:

```
int a;                  // external linkage
extern int b;           // external linkage
static int c;           // internal linkage

void foo() {
    int a;              // no linkage, automatic (stack)
    static int b;       // no linkage, static
    extern int c;       // internal linkage
    extern int d;       // external linkage
}

static int b;           // undefined (lint complains but cc doesn't)
extern int c;           // internal linkage
```

3.31 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 10.2.)

Answer: As discussed in Sidebar 10.1, the "private" part of a module header or class declaration provides information that, while not semantically part of the interface of the abstraction, is nonetheless required by the compiler in order to generate code for clients of the abstraction. There are two problems with this notion. First, it violates the spirit of information hiding, because it allows the writers of client code to see inside the abstraction, even if they can't *use* what they see. Second, it forces the creator of the abstraction to think about, and explicitly identify, the extra information needed by the compiler.

As described in Section 10.2.1 ("Making Do Without Module Headers"), Java and C# take an alternative approach, dispensing with headers altogether. The creator of an abstraction identifies

the semantically public aspects of the interface. The compiler examines the full implementation to find whatever else it needs.

4 Semantic Analysis

4.9 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

4.1 Basic results from automata theory tell us that the language $L = \mathtt{a}^n\mathtt{b}^n\mathtt{c}^n = \epsilon$, `abc`, `aabbcc`, `aaabbbccc`, ... is not context free. It can be captured, however, using an attribute grammar. Give an underlying CFG and a set of attribute rules that associates a Boolean attribute ok with the root R of each parse tree, such that R.ok = true if and only if the string corresponding to the fringe of the tree is in L.

Answer:

$G \longrightarrow As\ Bs\ Cs$	▷ G.ok := (As.val = Bs.val = Cs.val)
$As_1 \longrightarrow \mathtt{a}\ As_2$	▷ As_1.val := As_2.val + 1
$As \longrightarrow \epsilon$	▷ As.val := 0
$Bs_1 \longrightarrow \mathtt{b}\ Bs_2$	▷ Bs_1.val := Bs_2.val + 1
$Bs \longrightarrow \epsilon$	▷ Bs.val := 0
$Cs_1 \longrightarrow \mathtt{c}\ Cs_2$	▷ Cs_1.val := Cs_2.val + 1
$Cs \longrightarrow \epsilon$	▷ Cs.val := 0

4.2 Modify the grammar of Figure 2.25 so that it accepts only programs that contain at least one `write` statement. Make the same change in the solution to Exercise 2.17. Based on your experience, what do you think of the idea of using the CFG to enforce the rule that every function in C must contain at least one `return` statement?

Answer:

program ⟶ *other_stmt_list* `write` *expr stmt_list* `$$`
other_stmt_list ⟶ *other_stmt_list other_stmt* | ϵ
other_stmt ⟶ `id :=` *expr* | `read id`
stmt_list ⟶ *stmt_list stmt* | ϵ
stmt ⟶ `id :=` *expr* | `read id` | `write` *expr*
expr ⟶ *term* | *expr add_op term*
term ⟶ *factor* | *term mult_op factor*
factor ⟶ `(` *expr* `)` | `id` | `number`
add_op ⟶ `+` | `-`
mult_op ⟶ `*` | `/`

Here *other_stmt_list* generates all statements prior to the first `write` statement. Note that we have had to duplicate the two *stmt* productions other than `write`.

program ⟶ *other_stmt_list writing_stmt stmt_list* `$$`
other_stmt_list ⟶ *other_stmt_list other_stmt* | ϵ
other_stmt ⟶ `id :=` *expr* | `read id`
| `if` *condition* `then` *other_stmt_list* `fi`
| `while` *condition* `do` *other_stmt_list* `od`
writing_stmt ⟶ `write` *expr*
| `if` *condition* `then` *other_stmt_list writing_stmt stmt_list* `fi`
| `while` *condition* `do` *other_stmt_list writing_stmt stmt_list* `od`
stmt_list ⟶ *stmt_list stmt* | ϵ
stmt ⟶ `id :=` *expr* | `read id` | `write` *expr*
| `if` *condition* `then` *stmt_list* `fi`
| `while` *condition* `do` *stmt_list* `od`
condition ⟶ *expr relation expr*
relation ⟶ `<` | `>` | `<=` | `>=` | `=` | `!=`
expr ⟶ *term* | *expr add_op term*
term ⟶ *factor* | *term mult_op factor*
factor ⟶ `(` *expr* `)` | `id` | `number`
add_op ⟶ `+` | `-`
mult_op ⟶ `*` | `/`

This time there were *four* non-`write` *stmt* productions to duplicate. Worse, the *writing_stmt*, which is to contain the first `write` in the program, must have the same three-part inner structure as *program*. In a real programming language, there will typically be several more kinds of statements that may contain nested statements. To force the occurrence of a `return` statement without introducing ambiguity into the language, each of these would have to mimic the three-part inner structure of *program*. Making the mandatory `return` a matter of semantics rather than syntax is significantly simpler.

4.3 Give two examples of reasonable semantic rules that *cannot* be checked at reasonable cost, either statically or by compiler-generated code at run time.

Answer: In most programs (operating systems and some servers are counter-examples) all loops should terminate. Termination is undecidable, however. Similarly, most concurrent programs should be free from race conditions (Example 13.2), but this, too, is undecidable. Other conditions are in principle decidable, but so difficult to check that most language implementations don't try. Examples include improper use of variants (Section C-8.1.3), use of uninitialized variables (Section 6.1.3), deadlock freedom (Section 13.4.1), and the requirement in Ada that a program be unable to tell the difference between reference and value-result parameters (Section 9.3.1).

4.4 Write an S-attributed attribute grammar, based on the CFG of Example 4.7, that accumulates the value of the overall expression into the root of the tree. You will need to use dynamic memory allocation so that individual attributes can hold an arbitrary amount of information.

Answer:

expr ⟶ `const` *expr_tail*	▷ expr.v := reduce(const.v, expr_tail.l)
expr_tail ⟶ `-` `const` *expr_tail*	▷ expr_tail_1.l := cons(const.v, expr_tail_2.l)
expr_tail ⟶ ϵ	▷ expr_tail.l := null

where

reduce(val, list) = (if list = null then val else reduce(val−head(list), tail(list)))

and cons is defined as in the following question.

4.5 Lisp has the unusual property that its programs take the form of parenthesized lists. The natural syntax tree for a Lisp program is thus a tree of binary cells (known in Lisp as `cons` cells), where the first child represents the first element of the list and the second child represents the rest of the list. The syntax tree for `(cdr '(a b c))` appears in Figure 4.16. (The notation `'L` is syntactic sugar for `(quote L)`.)

Extend the CFG of Exercise 2.18 to create an attribute grammar that will build such trees. When a parse tree has been fully decorated, the root should have an attribute v that refers to the syntax tree. You may assume that each atom has a synthesized attribute v that refers to a syntax tree node that holds information from the scanner. In your semantic functions, you may assume the availability of a cons function that takes two references as arguments and returns a reference to a new `cons` cell containing those references.

Answer:

P ⟶ E `$$`	▷ P.v := E.v
E ⟶ `atom`	▷ E.v := atom.v
E_1 ⟶ `'` E_2	▷ E_1.v := cons(quote, cons(E_2.v, null))
E_1 ⟶ `(` E_2 Es `)`	▷ E_1.v := cons(E_2.v, Es.v)
Es_1 ⟶ E Es_2	▷ Es_1.v := cons(E.v, Es_2.v)
Es ⟶ ϵ	▷ Es.v := null

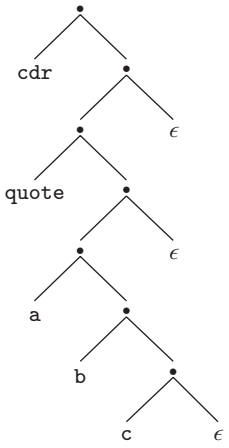


Figure 4.16 Natural syntax tree for the Lisp expression `(cdr '(a b c))`.

4.6 Refer back to the context-free grammar of Exercise 2.13. Add attribute rules to the grammar to accumulate into the root of the tree a count of the maximum depth to which parentheses are nested in the program string. For example, given the string `f1(a, f2(b * (c + (d - (e - f)))))`, the *stmt* at the root of the tree should have an attribute with a count of 3 (the parentheses surrounding argument lists don't count).

Answer:

stmt ⟶ *assignment*	▷ stmt.depth := assignment.depth
stmt ⟶ *subr_call*	▷ stmt.depth := subr_call.depth
assignment ⟶ `id :=` *expr*	▷ assignment.depth := expr.depth
subr_call ⟶ `id (` *arg_list* `)`	▷ subr_call.depth := arg_list.depth
expr ⟶ *primary expr_tail*	▷ expr.depth := max(primary.depth, expr_tail.depth)
expr_tail ⟶ *op expr*	▷ expr_tail.depth := expr.depth
expr_tail ⟶ ϵ	▷ expr_tail.depth := 0
primary ⟶ `id`	▷ primary.depth := 0
primary ⟶ *subr_call*	▷ primary.depth := subr_call.depth
primary ⟶ `(` *expr* `)`	▷ primary.depth := expr.depth + 1
op ⟶ `+` | `-` | `*` | `/`	
arg_list ⟶ *expr args_tail*	▷ args_list.depth := max(expr.depth, args_tail.depth)
args_tail ⟶ `,` *arg_list*	▷ args_tail.depth := args_list.depth
args_tail ⟶ ϵ	▷ args_tail.depth := 0

4.7 Suppose that we want to translate constant expressions into the postfix, or "reverse Polish" notation of logician Jan Łukasiewicz. Postfix notation does not require parentheses. It appears in stack-based languages such as Postscript, Forth, and the P-code and Java bytecode intermediate forms mentioned in Section 1.4. It also served, historically, as the input language of certain hand-held calculators made by Hewlett-Packard. When given a number, a postfix calculator would push the number onto an internal stack. When given an operator, it would pop the top two numbers from the stack, apply the operator, and push the result. The display would show the value at the top of the stack. To compute $2 \times (15 - 3)/4$, for example, one would push `2 E 1 5 E 3 E - * 4 E /` (here E is the "enter" key, used to end the string of digits that constitute a number).

Using the underlying CFG of Figure 4.1, write an attribute grammar that will associate with the root of the parse tree a sequence of postfix calculator button pushes, seq, that will compute the arithmetic value of the tokens derived from that symbol. You may assume the existence of a function buttons(c) that returns a sequence of button pushes (ending with E on a postfix calculator) for the constant `c`. You may also assume the existence of a concatenation function for sequences of button pushes.

Answer:

$E_1 \longrightarrow E_2$ `+` T	▷ E_1.seq := cat(E_2.seq, T.seq, "+")
$E_1 \longrightarrow E_2$ `-` T	▷ E_1.seq := cat(E_2.seq, T.seq, "−")
$E \longrightarrow T$	▷ E.seq := T.seq
$T_1 \longrightarrow T_2$ `*` F	▷ T_1.seq := cat(T_2.seq, F.seq, "×")
$T_1 \longrightarrow T_2$ `/` F	▷ T_1.seq := cat(T_2.seq, F.seq, "÷")
$T \longrightarrow F$	▷ T.seq := F.seq
$F_1 \longrightarrow$ `-` F_2	▷ F_1.seq := cat(F_2.seq, "+/−")
$F \longrightarrow$ `(` E `)`	▷ F.seq := E.seq
$F \longrightarrow$ `const`	▷ F.seq := buttons(const.val)

4.8 Repeat the previous exercise using the underlying CFG of Figure 4.3.

Answer:

$E \longrightarrow T\ TT$
▷ TT.st := T.seq ▷ E.seq := TT.seq

$TT_1 \longrightarrow$ `+` $T\ TT_2$
▷ TT_2.st := cat(TT_1.st, T.seq, "+") ▷ TT_1.seq := TT_2.seq

$TT_1 \longrightarrow$ `-` $T\ TT_2$
▷ TT_2.st := cat(TT_1.st, T.seq, "−") ▷ TT_1.seq := TT_2.seq

$TT \longrightarrow \epsilon$
▷ TT.seq := TT.st

$T \longrightarrow F\ FT$
▷ FT.st := F.seq ▷ T.seq := FT.seq

$FT_1 \longrightarrow$ `*` $F\ FT_2$
▷ FT_2.st := cat(FT_1.st, F.seq, "×") ▷ FT_1.seq := FT_2.seq

$FT_1 \longrightarrow$ `/` $F\ FT_2$
  ▷ FT_2.st := cat(FT_1.st, F.seq, "÷") ▷ FT_1.seq := FT_2.seq

$FT \longrightarrow \epsilon$
  ▷ FT.seq := FT.st

$F_1 \longrightarrow$ `-` F_2
  ▷ F_1.seq := cat(F_2.seq, "+/_")

$F \longrightarrow$ `(` E `)`
  ▷ F.seq := E.seq

$F \longrightarrow$ `const`
  ▷ F.seq := const.val

4.9 Consider the following grammar for reverse Polish arithmetic expressions:

$E \longrightarrow E\ E\ op$ | `id`
$op \longrightarrow$ `+` | `-` | `*` | `/`

Assuming that each `id` has a synthesized attribute name of type string, and that each E and op has an attribute val of type string, write an attribute grammar that arranges for the val attribute of the root of the parse tree to contain a translation of the expression into conventional infix notation. For example, if the leaves of the tree, left to right, were "`A A B - * C /`," then the val field of the root would be "`( ( A * ( A - B ) ) / C )`." As an extra challenge, write a version of your attribute grammar that exploits the usual arithmetic precedence and associativity rules to use as few parentheses as possible.

Answer:

$E_1 \longrightarrow E_2\ E_3\ op$	▷ E_1.val := cat('(', E_2.val, op.val, E_3.val, ')')
$E \longrightarrow$ `id`	▷ E.val := id.name
$op \longrightarrow$ `+`	▷ op.val := '+'
$op \longrightarrow$ `-`	▷ op.val := '−'
$op \longrightarrow$ `*`	▷ op.val := '*'
$op \longrightarrow$ `/`	▷ op.val := '/'

To minimize parentheses, let us say that constants have precedence 3, products and quotients have precedence 2, and sums and differences have precedence 1. To override left-to-right associativity, we need to parenthesize a right operand if its outermost operator has the same precedence as the current operator. To enforce default precedence, we need to parenthesize either a right or left operand if its outermost operator has precedence lower than that of the current operator.

$E_1 \longrightarrow E_2\ E_3\ op$	▷ E_1.val := assemble(E_2.val, E_2.prec, E_3.val, E_3.prec, op.val, op.prec) ▷ E_1.prec := op.prec	
$E \longrightarrow$ `id`	▷ E.val := id.name	▷ E.prec := 3
$op \longrightarrow$ `+`	▷ op.val := '+'	▷ op.prec := 1
$op \longrightarrow$ `-`	▷ op.val := '−'	▷ op.prec := 1
$op \longrightarrow$ `*`	▷ op.val := '*'	▷ op.prec := 2
$op \longrightarrow$ `/`	▷ op.val := '/'	▷ op.prec := 2

where

```
assemble(Lv, Lp, Rv, Rp, Ov, Op) =
    (let L = (if O.p > L.p then cat('(', L.v, ')') else L.v),
         R = (if O.p ≥ R.p then cat('(', R.v, ')') else R.v)
     in cat(L, O.v, R))
```

4.10 To reduce the likelihood of typographic errors, the digits comprising most credit card numbers are designed to satisfy the so-called *Luhn formula*, standardized by ANSI in the 1960s, and named for IBM mathematician Hans Peter Luhn. Starting at the right, we double every other digit (the second-to-last, fourth-to-last, etc.). If the doubled value is 10 or more, we add the resulting digits. We then sum together all the digits. In any valid number the result will be a multiple of 10. For example, 1234 5678 9012 3456 becomes 2264 1658 9022 6416, which sums to 64, so this is not a valid number. If the last digit had been 2, however, the sum would have been 60, so the number would potentially be valid.

Give an attribute grammar for strings of digits that accumulates into the root of the parse tree a Boolean value indicating whether the string is valid according to Luhn's formula. Your grammar should accommodate strings of arbitrary length.

Answer:

$CCN \longrightarrow DS$ ▷ CCN.valid := (DS.sum $\equiv$ 0 mod 10)

$DS \longrightarrow$ `digit` ▷ DS.even := false
▷ DS.sum := digit.value

$DS_1 \longrightarrow$ `digit` DS_2 ▷ DS_1.even := not DS_2.even
▷ DS_1.sum := DS_2.sum +
(if DS_2.even then d.value
else (if d.value $<$ 5 then 2 $\times$ d.value
else 2 $\times$ d.value $-$ 9))

4.11 Consider the following CFG for floating-point constants, without exponential notation. (Note that this exercise is somewhat artificial: the language in question is regular, and would be handled by the scanner of a typical compiler.)

$C \longrightarrow digits$ `.` $digits$
$digits \longrightarrow digit\ more\_digits$
$more\_digits \longrightarrow digits \mid \epsilon$
$digit \longrightarrow$ `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9`

Augment this grammar with attribute rules that will accumulate the value of the constant into a val attribute of the root of the parse tree. Your answer should be S-attributed.

Answer:

$C \longrightarrow digits_1$ `.` $digits_2$
▷ C.val := digits_1.val + digits_2.val $\times 10^{-\text{digits}_2\text{.len}}$

$digits \longrightarrow digit\ more\_digits$

▷ digits.val := digit.val $\times 10^{\text{more\_digits.len}}$ + more_digits.val
▷ digits.len := more_digits.len + 1

more_digits $\longrightarrow$ *digits*
▷ more_digits.val := digits.val
▷ more_digits.len := digits.len

more_digits $\longrightarrow$ ϵ
▷ more_digits.val := 0
▷ more_digits.len := 0

digit $\longrightarrow$ `0` ▷ digit.val := 0
digit $\longrightarrow$ `1` ▷ digit.val := 1
digit $\longrightarrow$ `2` ▷ digit.val := 2
digit $\longrightarrow$ `3` ▷ digit.val := 3
digit $\longrightarrow$ `4` ▷ digit.val := 4
digit $\longrightarrow$ `5` ▷ digit.val := 5
digit $\longrightarrow$ `6` ▷ digit.val := 6
digit $\longrightarrow$ `7` ▷ digit.val := 7
digit $\longrightarrow$ `8` ▷ digit.val := 8
digit $\longrightarrow$ `9` ▷ digit.val := 9

4.12 One potential criticism of the obvious solution to the previous problem is that the values in internal nodes of the parse tree do not reflect the value, in context, of the fringe below them. Create an alternative solution that addresses this criticism. More specifically, create your grammar in such a way that the `val` of an internal node is the sum of the `val`s of its children. Illustrate your solution by drawing the parse tree and attribute flow for `12.34`. (Hint: You will probably want a different underlying CFG, and non-L-attributed flow.)

Answer:

C $\longrightarrow$ *L_digits* . *R_digits*
▷ R_digits.pos := −1
▷ C.val := L_digits.val + R_digits.val

L_digits $\longrightarrow$ *digit more_L_digits*
▷ digit.pos := more_L_digits.len
▷ L_digits.len := more_L_digits.len + 1
▷ L_digits.val := digit.val + more_L_digits.val

more_L_digits $\longrightarrow$ *L_digits*
▷ more_L_digits.len := L_digits.len
▷ more_L_digits.val := L_digits.val

more_L_digits $\longrightarrow$ ϵ
▷ more_L_digits.len := 0
▷ more_L_digits.val := 0

R_digits $\longrightarrow$ *digit more_R_digits*
▷ digit.pos := R_digits.pos
▷ more_R_digits.pos := R_digits.pos − 1
▷ R_digits.val := digit.val + more_R_digits.val

more_R_digits $\longrightarrow$ *R_digits*
$\triangleright$ R_digits.pos := more_R_digits.pos
$\triangleright$ more_R_digits.val := R_digits.val

more_R_digits $\longrightarrow$ ϵ
$\triangleright$ more_R_digits.val := 0

digit $\longrightarrow$ 0
$\triangleright$ digit.val := $0 \times 10^{\text{digit.pos}}$

digit $\longrightarrow$ 1
$\triangleright$ digit.val := $1 \times 10^{\text{digit.pos}}$

digit $\longrightarrow$ 2
$\triangleright$ digit.val := $2 \times 10^{\text{digit.pos}}$

digit $\longrightarrow$ 3
$\triangleright$ digit.val := $3 \times 10^{\text{digit.pos}}$

digit $\longrightarrow$ 4
$\triangleright$ digit.val := $4 \times 10^{\text{digit.pos}}$

digit $\longrightarrow$ 5
$\triangleright$ digit.val := $5 \times 10^{\text{digit.pos}}$

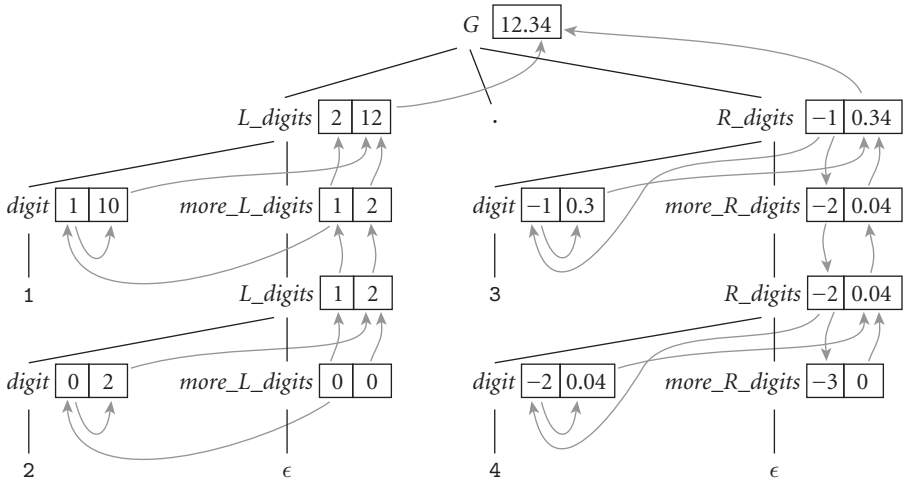
digit $\longrightarrow$ 6
$\triangleright$ digit.val := $6 \times 10^{\text{digit.pos}}$

digit $\longrightarrow$ 7
$\triangleright$ digit.val := $6 \times 10^{\text{digit.pos}}$

digit $\longrightarrow$ 8
$\triangleright$ digit.val := $8 \times 10^{\text{digit.pos}}$

digit $\longrightarrow$ 9
$\triangleright$ digit.val := $9 \times 10^{\text{digit.pos}}$

In the following, the double boxes on *digit*, *R_digits*, and *more_R_digits* represent pos (left) and val (right). The boxes on *L_digits* and *more_L_digits* represent len (left) and val (right). The single box for *G* is val.



4.13 Consider the following attribute grammar for variable declarations, based on the CFG of Exercise 2.11:

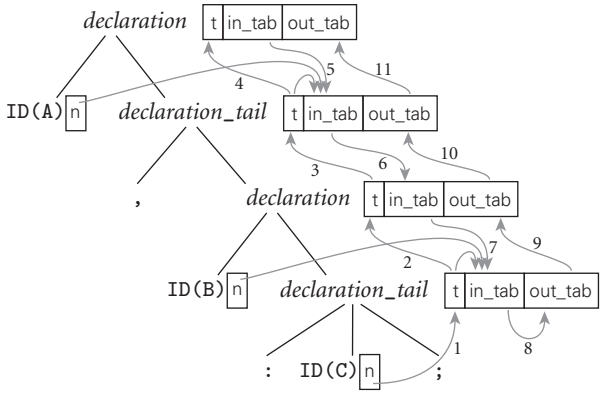
decl ⟶ `ID` *decl_tail*
- ▷ decl.t := decl_tail.t
- ▷ decl_tail.in_tab := insert (decl.in_tab, ID.n, decl_tail.t)
- ▷ decl.out_tab := decl_tail.out_tab

decl_tail ⟶ `,` *decl*
- ▷ decl_tail.t := decl.t
- ▷ decl.in_tab := decl_tail.in_tab
- ▷ decl_tail.out_tab := decl.out_tab

decl_tail ⟶ `: ID ;`
- ▷ decl_tail.t := ID.n
- ▷ decl_tail.out_tab := decl_tail.in_tab

Show a parse tree for the string `A, B : C;`. Then, using arrows and textual description, specify the attribute flow required to fully decorate the tree. (Hint: Note that the grammar is *not* L-attributed.)

Answer:



Decoration occurs in three phases. First the type propagates up to the top of the tree. Then the symbol table propagates down the tree, picking up declarations along the way. Finally the resulting symbol table propagates back up the tree.

4.14 A CFG-based attribute evaluator capable of handling non-L-attributed attribute flow needs to take a parse tree as input. Explain how to build a parse tree automatically during a top-down or bottom-up parse (i.e., without explicit action routines).

Answer: For a top-down parser, revise Figure 2.19 as follows.

```
terminal = 1 . . number_of_terminals
non_terminal = number_of_terminals + 1 . . number_of_symbols
symbol = 1 . . number_of_symbols
production = 1 . . number_of_productions
node = record
    parent : symbol
    children : list of node

parse_tab : array [non_terminal, terminal] of record
    action : (predict, error)
    prod : production
prod_tab : array [production] of list of symbol
-- these two tables are created by a parser generator tool

parse_stack : stack of node

root : node := new node(start_symbol, null)
parse_stack.push(root)
loop
    top_node : node := parse_stack.pop()
    expected_sym : symbol := top_node.parent
    if expected_sym ∈ terminal
        match(expected_sym)                -- as in Figure 2.17
        if expected_sym = $$ return root   -- success!
```

```
        else
            if parse_tab[expected_sym, input_token].action = error
                parse_error
            else
                prediction : production := parse_tab[expected_sym, input_token].prod
                foreach sym : symbol in reverse prod_tab[prediction]
                    child : node := new node(sym, null)
                    parse_stack.push(child)
                    top_node.children.insert(child)
```

For a bottom-up parser, revise Figure 2.29 as follows.

```
state = 1 . . number_of_states
symbol = 1 . . number_of_symbols
production = 1 . . number_of_productions
action_rec = record
    action : (shift, reduce, shift_reduce, error)
    new_state : state
    prod : production
node = record
    parent : symbol
    children : list of node

parse_tab : array [symbol, state] of action_rec
prod_tab : array [production] of record
    lhs : symbol
    rhs_len : integer
-- these two tables are created by a parser generator tool

parse_stack : stack of record
    n : node
    st : state

parse_stack.push(⟨null, start_state⟩)
cur_sym : symbol := scan()                      -- get new token from scanner
cur_node : node := new node(cur_sym, null)
loop
    cur_state : state := parse_stack.top.st     -- peek at state at top of stack
    if cur_state = start_state and cur_sym = start_symbol
        return cur_node                         -- success!
    ar : action_rec := parse_tab[cur_state, cur_sym]
    case ar.action
        shift:
            parse_stack.push(cur_node, ar.new_state⟩)
            cur_sym := scan()                   -- get new token from scanner
            cur_node := new node(cur_sym, null)
        reduce:
            cur_sym := prod_tab[ar.prod].lhs
            cur_node := new node(cur_sym, null)
```

```
            repeat prod_tab[ar.prod].rhs_len times
                ⟨child : node, s : state⟩ := parse_stack.pop()
                cur_node.children.insert(child)
        shift_reduce:
            child : node := cur_node
            cur_sym := prod_tab[ar.prod].lhs
            cur_node := new node(cur_sym, [child])
            repeat prod_tab[ar.prod].rhs_len-1 times
                ⟨child, s : state⟩ := parse_stack.pop()
                cur_node.children.insert(child)
        error:
            parse_error
```

4.15 Building on Example 4.13, modify the remainder of the recursive descent parser of Figure 2.17 to build syntax trees for programs in the calculator language.

Answer: Here's code in C++. The `print` methods exist solely to verify that the tree has been built correctly. Given our sum-and-average program (Example 2.24), the parser would print `( read A ) ( read B ) ( sum := ( A + B ) ) ( write sum) ( write ( sum / 2 ) ).`

```
/* Complete recursive descent parser for the calculator language
    (solution to Exercise 2.21), augmented to build a syntax tree, as
    suggested in Figure 4.10.  Builds on figure 2.16.  Uses the scanner
    from problem 2.5 (lightly modified).  Does no error recovery.
    NB: This code has been augmented to include unary minus, which was
    not a part of the original calculator language, but does appear in
    Example 4.11.
*/

#include <iostream>
using std::cout;
#include <string>
using std::string;

#include "scan.h"

static token input_token;

string match(token expected);       // forward declaration

//---------------------------------------
//  syntax tree classes

struct stmt_n;  struct expr_n;      // forward declarations

struct program_n {
    stmt_n* start;
    program_n(stmt_n* s) : start(s) { }
    virtual void print();
};
```

```
struct stmt_n {
    stmt_n* next;
    stmt_n() { next = 0; }
    virtual void print() = 0;       // pure virtual base class
};

struct assign_n : public stmt_n {
    string lhs;
    expr_n* rhs;
    assign_n(string l, expr_n* r) : lhs(l), rhs(r) { }
    virtual void print();
};

struct read_n : public stmt_n {
    string target;
    read_n(string t) : target(t) { }
    virtual void print();
};

struct write_n : public stmt_n {
    expr_n* output;
    write_n(expr_n* e) : output(e) { }
    virtual void print();
};

struct expr_n {
    virtual void print() = 0;       // pure virtual base class
};

struct id_n : public expr_n {
    string name;
    id_n(string n) : name(n) { }
    virtual void print();
};

struct num_n : public expr_n {
    string val;
    num_n(string v) : val(v) { }
    virtual void print();
};

struct un_op_n : public expr_n {
    string op;
    expr_n* child;
    un_op_n(string o, expr_n* c) : op(o), child(c) { }
    virtual void print();
};
```

```
struct bin_op_n : public expr_n {
    string op;
    expr_n* left;
    expr_n* right;
    bin_op_n(string o, expr_n* l, expr_n* r) : op(o), left(l), right(r) { }
    virtual void print();
};

void program_n::print() {
    if (start) start->print();
}
void assign_n::print() {
    cout << "( " << lhs;
    cout << " := ";
    rhs->print();
    cout << ") ";
    if (next) next->print();
}
void read_n::print() {
    cout << "( read " << target << " ) ";
    if (next) next->print();
}
void write_n::print() {
    cout << "( write ";
    output->print();
    cout << ") ";
    if (next) next->print();
}
void id_n::print() {
    cout << name;
}
void num_n::print() {
    cout << val;
}
void un_op_n::print() {
    cout << op;
    child->print();
    cout << " ";
}
void bin_op_n::print() {
    cout << "( ";
    left->print();
    cout << " " << op << " ";
```

```
    right->print();
    cout << " ) ";
}
//----------------------------------------

string match(token expected) {
    if (input_token == expected) {
        string tok = string(token_image);
        input_token = scan();
        return tok;
    }
    else error();
}

program_n* program();                   // forward declarations
stmt_n* stmt_list();
stmt_n* stmt();
expr_n* expr();
expr_n* term_tail(expr_n* e1);
expr_n* term();
expr_n* factor_tail(expr_n* e1);
expr_n* factor();
string add_op();
string mult_op();

program_n* program() {
    stmt_n* s;
    switch (input_token) {
        case tk_id:
        case tk_read:
        case tk_write:
        case tk_eof:
            s = stmt_list();
            (void) match(tk_eof);
            return new program_n(s);
        default: error();
    }
}

stmt_n* stmt_list() {
    stmt_n* s;
    stmt_n* t;
    switch (input_token) {
        case tk_id:
        case tk_read:
```

```
        case tk_write:
            s = stmt();
            t = stmt_list();
            s->next = t;
            return s;
        case tk_eof:
            return 0;       /* epsilon production */
        default: error();
    }
}

stmt_n* stmt() {
    string lhs;
    expr_n* rhs;
    expr_n* output;
    string target;
    switch (input_token) {
        case tk_id:
            lhs = match(tk_id);
            (void) match(tk_becomes);
            rhs = expr();
            return new assign_n(lhs, rhs);
        case tk_read:
            (void) match(tk_read);
            target = match(tk_id);
            return new read_n(target);
        case tk_write:
            (void) match(tk_write);
            output = expr();
            return new write_n(output);
        default: error();
    }
}

expr_n* expr() {
    expr_n* e;
    switch (input_token) {
        case tk_id:
        case tk_literal:
        case tk_lparen:
        case tk_sub:
            e = term();
            return term_tail(e);
        default: error();
    }
}
```

```
expr_n* term_tail(expr_n* e1) {
    string o;
    expr_n* e2;
    switch (input_token) {
        case tk_add:
        case tk_sub:
            o = add_op();
            e2 = term();
            return term_tail(new bin_op_n(o, e1, e2));
        case tk_rparen:
        case tk_id:
        case tk_read:
        case tk_write:
        case tk_eof:
            return e1;      /*  epsilon production */
        default: error();
    }
}

expr_n* term() {
    expr_n* f;
    switch (input_token) {
        case tk_id:
        case tk_literal:
        case tk_lparen:
        case tk_sub:
            f = factor();
            return factor_tail(f);
        default: error();
    }
}

expr_n* factor_tail(expr_n *e1) {
    string o;
    expr_n* e2;
    switch (input_token) {
        case tk_mul:
        case tk_div:
            o = mult_op();
            e2 = factor();
            return factor_tail(new bin_op_n(o, e1, e2));
        case tk_add:
        case tk_sub:
        case tk_rparen:
        case tk_id:
        case tk_read:
        case tk_write:
```

```
        case tk_eof:
            return e1;      /*  epsilon production */
        default: error();
    }
}

expr_n* factor() {
    string o;
    expr_n* e;
    switch (input_token) {
        case tk_id :
            return new id_n(match(tk_id));
        case tk_literal:
            return new num_n(match(tk_literal));
        case tk_sub:
            o = match(tk_sub);
            e = expr();
            return new un_op_n(o, e);
        case tk_lparen:
            (void) match(tk_lparen);
            e = expr();
            (void) match(tk_rparen);
            return e;
        default: error();
    }
}

string add_op() {
    switch (input_token) {
        case tk_add:
            return match(tk_add);
        case tk_sub:
            return match(tk_sub);
        default: error();
    }
}

string mult_op() {
    switch (input_token) {
        case tk_mul:
            return match(tk_mul);
        case tk_div:
            return match(tk_div);
        default: error();
    }
}
```

```
int main() {
    input_token = scan();
    program_n* p = program();
    p->print();
    cout << "\n";
}
```

4.16 Write an LL(1) grammar with action routines and automatic attribute space management that generates the reverse Polish translation described in Exercise 4.7.

Answer:

$E \longrightarrow T$ { TT.st := T.seq } TT { E.seq := TT.seq }
$TT_1 \longrightarrow$ `+` T { TT_2.st := cat(TT_1.st, T.seq, "+") } TT_2 { TT_1.seq := TT_2.seq }
$TT_1 \longrightarrow$ `-` T { TT_2.st := cat(TT_1.st, T.seq, "−") } TT_2 { TT_1.seq := TT_2.seq }
$TT \longrightarrow \epsilon$ { TT.seq := TT.st }
$T \longrightarrow F$ { FT.st := F.seq } FT { T.seq := FT.seq }
$FT_1 \longrightarrow$ `*` F { FT_2.st := cat(FT_1.st, F.seq, "×") } FT_2 { FT_1.seq := FT_2.seq }
$FT_1 \longrightarrow$ `/` F { FT_2.st := cat(FT_1.st, F.seq, "÷") } FT_2 { FT_1.seq := FT_2.seq }
$FT \longrightarrow \epsilon$ { FT.seq := FT.st }
$F_1 \longrightarrow$ `-` F_2 { F_1.seq := cat(F_2.seq, "+/−") }
$F \longrightarrow$ `(` E `)` { F.seq := E.seq }
$F \longrightarrow$ `const` { F.seq := const.val }

4.17 (a) Write a context-free grammar for polynomials in x. Add semantic functions to produce an attribute grammar that will accumulate the polynomial's derivative (as a string) in a synthesized attribute of the root of the parse tree.

Answer: The following assumes that exponents in the input are all positive integers.

$P \longrightarrow T$ *more_Ts*
  ▷ more_Ts.st := T.d ▷ P.d := more_Ts.d
$T \longrightarrow$ `num` *T_tail*
  ▷ T_tail.c := num.v ▷ T.d := T_tail.d
T_tail $\longrightarrow$ `x` *exp*
  ▷ exp.c := T_tail.c ▷ T_tail.d := exp.d
T_tail $\longrightarrow \epsilon$
  ▷ T_tail.d := " "
exp $\longrightarrow$ `**` `num`
  ▷ exp.d := float_to_string(exp.c × num.v) + "x **" + int_to_string(num.v − 1)
exp $\longrightarrow \epsilon$
  ▷ exp.d := float_to_string(exp.c)
more_Ts$_1 \longrightarrow$ `+` T *more_Ts*$_2$
  ▷ more_Ts$_2$.st := more_Ts$_1$.st + "+" + T.d ▷ more_Ts$_1$.d := more_Ts$_2$.d
more_Ts $\longrightarrow \epsilon$
  ▷ more_Ts.d := more_Ts.st

(b) Replace your semantic functions with action routines that can be evaluated during parsing.

Answer:

P ⟶ *T* { more_Ts.st := T.d } *more_Ts* { P.d := more_Ts.d }
T ⟶ `num` { T_tail.c := num.v } *T_tail* { T.d := T_tail.d }
T_tail ⟶ `x` { exp.c := T_tail.c } *exp* { T_tail.d := exp.d }
⟶ ϵ { T_tail.d := " " }
exp ⟶ `**` `num` { exp.d := float_to_string(exp.c × num.v)
+ "x **" + int_to_string(num.v − 1) }
⟶ ϵ { exp.d := float_to_string(exp.c) }
more_Ts$_1$ ⟶ `+` *T* { more_Ts$_2$.st := more_Ts$_1$.st + "+" + T.d } *more_Ts*$_2$
{ more_Ts$_1$.d := more_Ts$_2$.d }
⟶ ϵ { more_Ts.d := more_Ts.st }

4.18 (a) Write a context-free grammar for `case` or `switch` statements in the style of Pascal or C. Add semantic functions to ensure that the same label does not appear on two different arms of the construct.

Answer: In the following, the input is error-free if and only if S.dups = ∅.

S ⟶ `switch` `(` *expr* `)` `{` *arm_list* `}` ▷ S.dups := arm_list.dups
arm_list ⟶ *case_list stmt more_arms*
▷ arm_list.out := case_list.out ∪ more_arms.out
▷ arm_list.dups := case_list.dups ∪ more_arms.dups
∪ (case_list.out ∩ more_arms.out)
case_list ⟶ `case` *expr* `:` *more_cases*
▷ case_list.out := {expr.val} ∪ more_cases.out
▷ case_list.dups := more_cases.dups ∪ ({expr.val} ∩ more_cases.out)
more_cases ⟶ *case_list*
▷ more_cases.out := case_list.out ▷ more_cases.dups := case_list.dups
more_cases ⟶ ϵ
▷ more_cases.out := ∅ ▷ more_cases.dups := ∅
more_arms ⟶ *arm_list*
▷ more_arms.out := arm_list.out ▷ more_arms.dups := arm_list.dups
more_arms ⟶ ϵ
▷ more_arms.out := ∅ ▷ more_arms.dups := ∅

(b) Replace your semantic functions with action routines that can be evaluated during parsing.

Answer: The grammar in (a) is S-attributed, so this solution is particularly simple:

S ⟶ `switch` `(` *expr* `)` `{` *arm_list* `}` { S.dups := arm_list.dups }
arm_list ⟶ *case_list stmt more_arms*
{ arm_list.out := case_list.out ∪ more_arms.out;
arm_list.dups := case_list.dups ∪ more_arms.dups
∪ (case_list.out ∩ more_arms.out) }

case_list ⟶ `case` *expr* `:` *more_cases*
{ case_list.out := {expr.val} ∪ more_cases.out;
case_list.dups := more_cases.dups ∪ ({expr.val} ∩ more_cases.out) }

more_cases ⟶ *case_list*
{ more_cases.out := case_list.out; more_cases.dups := case_list.dups }

⟶ ϵ { more_cases.out := ∅; more_cases.dups := ∅ }

more_arms ⟶ *arm_list*
{ more_arms.out := arm_list.out; more_arms.dups := arm_list.dups }

⟶ ϵ { more_arms.out := ∅; more_arms.dups := ∅ }

4.19 Write an algorithm to determine whether the rules of an arbitrary attribute grammar are noncircular. (Your algorithm will require exponential time in the worst case [JOR75].)

Answer:

4.20 Rewrite the attribute grammar of Figure 4.14 in the form of an ad hoc tree traversal consisting of mutually recursive subroutines in your favorite programming language. Keep the symbol table in a global variable, rather than passing it through arguments.

Answer: Here's a solution in Python. We use classes to represent the various kinds of tree node. Constructor (`__init__`) arguments—tree links and location information—are assumed to be pre-initialized by the code that builds the syntax tree, before AG-based decoration begins. Because this is an L-atteributed AG, we can decorate the tree in a single left-to-right depth-first traversal, and we can get by without parent pointers, which it makes it easier to construct trees as a single expression in the examples. In the code below, inherited attributes are passed to the `decorate` method; synthesized attributes are returned from that method. One could also use fields (data members) of the nodes to represent attributes explicitly, and assign to them in the `decorate` method.

```
def declare_name(i, t, l):      # returns error message, if any
    if i in symtab:
        symtab[i] = "error"
        return "redefinition of " + i + " at " + l + "\n"
    else:
        symtab[i] = t
        return ""

def check_types(t1, t2, errors_below, l):
    # returns (type, e), where e is error message, if any
    if t1 == "error" or t2 == "error":
        return "error", errors_below
    elif t1 != t2:
        return "error", errors_below + "type clash at " + l + "\n"
    else:
        return t1, errors_below
```

```
class program:
    def __init__(self, l, m):
        self.location = str(l)
        self.main = m
    def decorate(self):             # returns errors
        global symtab
        symtab = {}                 # initially empty
        return self.main.decorate("")

class item:
    def __init__(self, l, n):
        self.location = str(l)
        self.next = n
    def decorate(self, errors_in):  # returns errors_out
        raise Exception("unexpected call to item.decorate")

class int_decl(item):
    def __init__(self, l, i, n):
        item.__init__(self, l, n)
        self.ident = i
    def decorate(self, errors_in):  # returns errors_out
        return self.next.decorate(errors_in +
            declare_name(self.ident, "int", self.location))

class real_decl(item):
    def __init__(self, l, i, n):
        item.__init__(self, l, n)
        self.ident = i
    def decorate(self, errors_in):  # returns errors_out
        return self.next.decorate(errors_in +
            declare_name(self.ident, "real", self.location))

class read(item):
    def __init__(self, l, i, n):
        item.__init__(self, l, n)
        self.ident = i
    def decorate(self, errors_in):  # returns errors_out
        new_err = ""
        if not self.ident in symtab:
            new_err = self.ident + " undefined at " + self.location + "\n"
        return self.next.decorate(errors_in + new_err)

class write(item):
    def __init__(self, l, e, n):
        item.__init__(self, l, n)
        self.value = e
```

```
    def decorate(self, errors_in):  # returns errors_out
        e_type, e_errors = self.value.decorate()
        return self.next.decorate(errors_in + e_errors)

class assign(item):
    def __init__(self, l, i, e, n):
        item.__init__(self, l, n)
        self.ident = i
        self.value = e
    def decorate(self, errors_in):  # returns errors_out
        e_type, e_errors = self.value.decorate()
        if self.ident in symtab:
            i_type = symtab[self.ident]
            if i_type != "error" and e_type != "error" and e_type != i_type:
                errors_in += "type clash at " + self.location + "\n"
            else:
                errors_in += e_errors
        else:
            errors_in += (self.ident + " undefined at " +
                self.location + "\n" + e_errors)
        return self.next.decorate(errors_in)

class null(item):
    def __init__(self):
        pass
    def decorate(self, errors_in):  # returns errors_out
        return errors_in

class expr:
    def __init__(self, l):
        self.location = str(l)
    def decorate(self):             # returns (type, errors)
        raise Exception("unexpected call to expr.decorate")

class id(expr):
    def __init__(self, l, i):
        expr.__init__(self, l)
        self.ident = i
    def decorate(self):             # returns (type, errors)
        if self.ident in symtab:
            return symtab[self.ident], ""
        else:
            return "error", (self.ident + " undefined at " +
                self.location + "\n")
```

```
class int_const(expr):
    def __init__(self, l, v):
        expr.__init__(self, l)
        self.value = v
    def decorate(self):             # returns (type, errors)
        return "int", ""

class real_const(expr):
    def __init__(self, l, v):
        expr.__init__(self, l)
        self.value = v
    def decorate(self):             # returns (type, errors)
        return "real", ""

def bin_op_init(self, l, e1, e2):
    expr.__init__(self, l)
    self.left = e1
    self.right = e2

def bin_op_decorate(self):
    t1, err1 = self.left.decorate()
    t2, err2 = self.right.decorate()
    return check_types(t1, t2, err1 + err2, self.location)

class plus(expr):
    __init__ = bin_op_init
    decorate = bin_op_decorate

class minus(expr):
    __init__ = bin_op_init
    decorate = bin_op_decorate

class times(expr):
    __init__ = bin_op_init
    decorate = bin_op_decorate

class div(expr):
    __init__ = bin_op_init
    decorate = bin_op_decorate

class float(expr):
    def __init__(self, l, e):
        expr.__init__(self, l)
        self.operand = e
```

```
    def decorate(self):             # returns (type, errors)
        old_type, errors_below = self.operand.decorate()
        if old_type == "int" or old_type == "error":
            return "real", errors_below
        else:
            return "error", (errors_below +
                "float of non-int at " + self.location)

class trunc(expr):
    def __init__(self, e):
        expr.__init__(self, l)
        self.operand = e
    def decorate(self):             # returns (type, errors)
        old_type, errors_below = self.e.decorate()
        if old_type == "real" or old_type == "error":
            return "int", errors_below
        else:
            return "error", (errors_below +
                "trunc of non-real at " + self.location)

####################
# testing:

tree1 = program(1,                  # Fig. 4.12
                int_decl(1, "a",
                read(2, "a",
                real_decl(3, "b",
                read(4, "b",
                write(5,
                      div(5,
                          plus(5,
                               float(5, id(5, "a")),
                               id(5, "b")),
                          real_const(5, 2.0)),
                null()))))))
print("program errors: [" + tree1.decorate() + "]")

tree2 = program(1,
                int_decl(1, "a",
                read(2, "a",
                real_decl(3, "b",
                read(4, "b",
```

```
                write(5,
                      div(5,
                          plus(5,
                               id(5, "a"),  # missing float
                               id(5, "b")),
                          real_const(5, 2.0)),
                null()))))))
print("program errors: [" + tree2.decorate() + "]")

tree3 = program(1,
                int_decl(1, "a",
                read(2, "a",
                read(4, "b",                # missing declaration of b
                write(5,
                      div(5,
                          plus(5,
                               float(5, id(5, "a")),
                               id(5, "b")),
                          real_const(5, 2.0)),
                null()))))))
print("program errors: [" + tree3.decorate() + "]")

tree4 = program(1,
                int_decl(1, "a",
                read(2, "a",
                real_decl(3, "b",
                read(4, "b",
                write(5,
                      div(5,
                          plus(5,
                               float(5, id(5, "a")),
                               id(5, "b")),
                          real_const(5, 2.0)),
                int_decl(6, "a",
                null()))))))))
print("program errors: [" + tree4.decorate() + "]")

tree5 = program(1,
                int_decl(1, "a",
                read(2, "a",
                real_decl(3, "b",
                read(4, "b",
```

```
                assign(5, "a",              # type clash
                       div(5,
                           plus(5,
                                float(5, id(5, "a")),
                                id(5, "b")),
                           real_const(5, 2.0)),
                null()))))))
print("program errors: [" + tree5.decorate() + "]")

tree6 = program(1,
                real_decl(3, "b",
                read(4, "b",
                assign(5, "a",              # missing declaration of a
                       div(5,
                           plus(5,
                                float(5, id(5, "a")),
                                id(5, "b")),
                           real_const(5, 2.0)),
                null()))))
print("program errors: [" + tree6.decorate() + "]")
```

4.21 Write an attribute grammar based on the CFG of Figure 4.11 that will build a syntax tree with the structure described in Figure 4.14.

Answer:

4.22 Augment the attribute grammar of Figure 4.5, Figure 4.6, or Exercise 4.21 to initialize a synthesized attribute in every syntax tree node that indicates the location (line and column) at which the corresponding construct appears in the source program. You may assume that the scanner initializes the location of every token.

Answer:

4.23 Modify the CFG and attribute grammar of Figures 4.11 and 4.14 to permit mixed integer and real expressions, without the need for `float` and `trunc`. You will want to add an annotation to any node that must be coerced to the opposite type, so that the code generator will know to generate code to do so. Be sure to think carefully about your coercion rules. In the expression `my_int + my_real`, for example, how will you know whether to coerce the integer to be a real, or to coerce the real to be an integer?

Answer:

4.24 Explain the need for the $A : B$ notation on the left-hand sides of productions in a tree grammar. Why isn't similar notation required for context-free grammars?

Answer: As noted in Example 4.16, the $A : B$ notation indicates that A is one variant of B, and may appear anywhere a B is expected on a right-hand side. We could, as we do in CFGs, introduce extra rules (e.g., *item* $\longrightarrow$ *int_decl*, *item* $\longrightarrow$ *read*, etc.), but these would need to correspond to extra nodes in the tree, which aren't there. The extra rules are *needed* in a CFG

in order to drive the discovery of structure (and they lead to large numbers of internal nodes in parse trees). By contrast, the structure of a already-existing syntax tree is unambiguous and self-evident.

4.25 A potential objection to the tree attribute grammar of Example 4.17 is that it repeatedly copies the entire symbol table from one node to another. In this particular tiny language, it is easy to see that the referencing environment never shrinks: the symbol table changes only with the addition of new identifiers. Exploiting this observation, show how to modify the pseudocode of Figure 4.14 so that it copies only pointers, rather than the entire symbol table.

Answer: The key idea is to represent the symbol table as a stack—specifically, a linked list onto which we push new elements at the front. All productions and attribute rules of Figure 4.14 can remain the same, provided we interpret ⟨name, ?⟩ ∈ symtab as a list-search operation that returns the first ⟨n, t⟩ pair in the table (if any) for which n = name. The declare_name macro requires only slightly more modification:

```
macro declare_name(id, cur_item, next_item : syntax_tree_node; t : type)
    if ⟨id.name, ?⟩ ∈ cur_item.symtab
        next_item.errors_in := cur_item.errors_in
            + ["redefinition of" id.name "at" cur_item.location]
        next_item.symtab := prepend(⟨id.name, error⟩, cur_item.symtab)
    else
        next_item.errors_in := cur_item.errors_in
        next_item.symtab := prepend(⟨id.name, t⟩, cur_item.symtab)
```

A similar mechanism would allow the various errors attributes to be implemented as pointers into a single, ever-growing list.

4.26 Your solution to the previous exercise probably doesn't generalize to languages with nontrivial scoping rules. Explain how an AG such as that in Figure 4.14 might be modified to use a global symbol table similar to the one described in Section C-3.4.1. Among other things, you should consider nested scopes, the hiding of names in outer scopes, and the requirement (not enforced by the table of Section C-3.4.1) that variables be declared before they are used.

Answer: The key idea is to include an indication of scope (referencing environment) in each node of the tree. For a language with static scope, the symbol table might consist of a "scope tree" and a mapping from ⟨id, scope_num⟩ pairs to type and other information. Each symtab attribute in the AST would be a pointer to a node of the symbol table scope tree. A lookup (⟨name, t⟩ ∈ symtab) operation would walk the path from the current scope node to the root of the scope tree. For each scope s on that path, in order, it would search for ⟨name, s⟩ in the hash table, stopping the first time it finds a match and returning the t to which ⟨name, s⟩ is mapped. To enforce declare-before-use, we might assign a serial number to each declaration in a scope, and indicate, in each syntax tree node, the largest serial number that precedes the construct rooted at that node. After identifying the matching declaration, the lookup routine would complain if the use of a name precedes its declaration.

IN MORE DEPTH

4.27 Repeat Exercise 4.7 using ad hoc attribute space management. Instead of accumulating the translation into a data structure, write it to a file on the fly.

Answer:

4.28 Rewrite the grammar for declarations of Example C-4.28 without the requirement that your attribute flow be L-attributed. Try to make the grammar as simple and elegant as possible (you shouldn't need to accumulate lists of identifiers).

Answer: Our solution takes the form of a true attribute grammar, rather than a CFG with action routines. Non-terminals *dec* and *id_list* carry a pair of attributes old_st and new_st that hold the compiler's symbol table before and after, respectively, elaboration of the declarations below them in the tree.

dec ⟶ *id_list* `:` `ID` `;`
- ▷ id_list.t := ID.n
- ▷ id_list.old_st := dec.old_st
- ▷ dec.new_st := id_list.new_st

id_list ⟶ `ID` *more_ids*
- ▷ more_ids.old_st := declare_variable (id_list.old_st, ID.n, id_list.t)
- ▷ more_ids.t := id_list.t
- ▷ id_list.new_st := more_ids.new_st

more_ids ⟶ `,` *id_list*
- ▷ id_list.t := more_ids.t
- ▷ id_list.old_st := more_ids.old_st
- ▷ more_ids.new_st := id_list.new_st

more_ids ⟶ ϵ
- ▷ more_ids.new_st := more_ids.old_st

4.29 Fill in the missing lines in Figure C-4.19.

Answer:

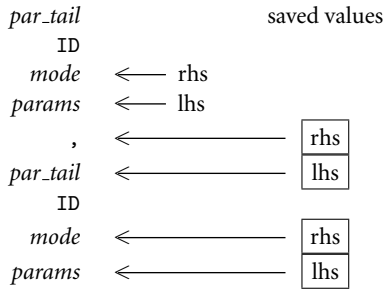
17:8FT**9:3**TT**4:2:) 16:8**FT**9:1**TT**2:**$	$E_? \; T_? \; TT_{?,?} \; F_? \; FT_{?,?} \; (\; E_? \;) \; T_1 \; TT_{1,?} + T_? \; TT_{?,?} \; \boxed{L} F_? \; FT_{?,?} \; \boxed{R} C_3 \; \boxed{N}$
:8FT**9:3**TT**4:2:) 16:8**FT**9:1**TT**2:**$	$E_? \; T_? \; TT_{?,?} \; F_? \; FT_{?,?} \; (\; E_? \;) \; T_1 \; TT_{1,?} + T_? \; TT_{?,?} \; \boxed{L} F_3 \; FT_{?,?} \; \boxed{R} C_3 \; \boxed{N}$
8FT**9:3**TT**4:2:) 16:8**FT**9:1**TT**2:**$	$E_? \; T_? \; TT_{?,?} \; F_? \; FT_{?,?} \; (\; E_? \;) \; T_1 \; TT_{1,?} + \boxed{L} T_? \; TT_{?,?} \; \boxed{R} F_3 \; \boxed{N} FT_{?,?}$
FT**9:3**TT**4:2:) 16:8**FT**9:1**TT**2:**$	$E_? \; T_? \; TT_{?,?} \; F_? \; FT_{?,?} \; (\; E_? \;) \; T_1 \; TT_{1,?} + \boxed{L} T_? \; TT_{?,?} \; \boxed{R} F_3 \; \boxed{N} FT_{3,?}$
14:9:3TT**4:2:) 16:8**FT**9:1**TT**2:**$	$E_? \; T_? \; TT_{?,?} \; F_? \; FT_{?,?} \; (\; E_? \;) \; T_1 \; TT_{1,?} + T_? \; TT_{?,?} \; F_3 \; \boxed{L} FT_{3,?} \; \boxed{R} \; \boxed{N}$
:9:3TT**4:2:) 16:8**FT**9:1**TT**2:**$	$E_? \; T_? \; TT_{?,?} \; F_? \; FT_{?,?} \; (\; E_? \;) \; T_1 \; TT_{1,?} + T_? \; TT_{?,?} \; F_3 \; \boxed{L} FT_{3,3} \; \boxed{R} \; \boxed{N}$
9:3TT**4:2:) 16:8**FT**9:1**TT**2:**$	$E_? \; T_? \; TT_{?,?} \; F_? \; FT_{?,?} \; (\; E_? \;) \; T_1 \; TT_{1,?} + \boxed{L} T_? \; TT_{?,?} \; \boxed{R} F_3 \; FT_{3,3} \; \boxed{N}$
:3TT**4:2:) 16:8**FT**9:1**TT**2:**$	$E_? \; T_? \; TT_{?,?} \; F_? \; FT_{?,?} \; (\; E_? \;) \; T_1 \; TT_{1,?} + \boxed{L} T_3 \; TT_{?,?} \; \boxed{R} F_3 \; FT_{3,3} \; \boxed{N}$
14:11:9:1TT**2:**$	$E_? \; T_? \; TT_{?,?} \; F_4 \; FT_{4,?} \; * \; F_2 \; \boxed{L} FT_{8,?} \; \boxed{R} \; \boxed{N}$
:11:9:1TT**2:**$	$E_? \; T_? \; TT_{?,?} \; F_4 \; FT_{4,?} \; * \; F_2 \; \boxed{L} FT_{8,8} \; \boxed{R} \; \boxed{N}$
11:9:1TT**2:**$	$E_? \; T_? \; TT_{?,?} \; F_4 \; \boxed{L} FT_{4,?} \; \boxed{R} * \; F_2 \; FT_{8,8} \; \boxed{N}$
:9:1TT**2:**$	$E_? \; T_? \; TT_{?,?} \; F_4 \; \boxed{L} FT_{8,8} \; \boxed{R} * \; F_2 \; FT_{8,8} \; \boxed{N}$
9:1TT**2:**$	$E_? \; \boxed{L} T_? \; TT_{?,?} \; \boxed{R} F_? \; FT_{8,8} \; \boxed{N}$
:1TT**2:**$	$E_? \; \boxed{L} T_8 \; TT_{?,?} \; \boxed{R} F_? \; FT_{8,8} \; \boxed{N}$

4.30 Consider the following grammar with action routines:

params ⟶ *mode* `ID` *par_tail*
{ params.list := insert(⟨mode.val, ID.name⟩, par_tail.list) }
par_tail ⟶ `,` *params* { par_tail.list := params.list }
⟶ { par_tail.list := null }
mode ⟶ `IN` { mode.val := IN }
⟶ `OUT` { mode.val := OUT }
⟶ `IN OUT` { mode.val := IN_OUT }

Suppose we are parsing the input `IN a, OUT b`, and that our compiler uses an automatically maintained attribute stack to hold the active slice of the parse tree. Show the contents of this attribute stack immediately before the parser predicts the production *par_tail* ⟶ ϵ. Be sure to indicate where $\boxed{\text{L}}\rangle$ and $\boxed{\text{R}}\rangle$ point in the attribute stack. Also show the stack of saved $\boxed{\text{L}}\rangle$ and $\boxed{\text{R}}\rangle$ values, showing where each points in the attribute stack. You may ignore the $\boxed{\text{N}}\rangle$ pointer.

Answer:



4.31 One problem with automatic space management for attributes in a top-down parser occurs in lists and sequences. Consider for example the following grammar:

block ⟶ `begin` *stmt_list* `end`
stmt_list ⟶ *stmt* *stmt_list_tail*
stmt_list_tail ⟶ `;` *stmt_list* | ϵ
stmt ⟶ ...

After predicting the final statement of an n-statement block, the attribute stack will contain the following (line breaks and indentation are for clarity only):

block begin stmt_list end
stmt stmt_list_tail ; stmt_list
stmt stmt_list_tail ; stmt_list
stmt stmt_list_tail ; stmt_list
{ n times }

If the attribute stack is of finite size, it is guaranteed to overflow for some long but valid block of straight-line code. The problem is especially unfortunate since,

with the exception of the accumulated output code, none of the repeated symbols in the attribute stack contains any useful attributes once its substructure has been parsed.

Suggest a technique to "squeeze out" useless symbols in the attribute stack, dynamically. Ideally, your technique should be amenable to automatic implementation, so it does not constitute a burden on the compiler writer.

Also, suppose you are using a compiler with a top-down parser that employs an automatically managed attribute stack, but does not squeeze out useless symbols. What could you do if your program caused the compiler to run out of stack space? How could you modify your program to "get around" the problem?

Answer:

5 Target Machine Architecture

5.8 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

IN MORE DEPTH

5.1 Consider sending a message containing a string of integers over the Internet. What problems may occur if the sending and receiving machines have different "endian-ness"? How might you solve these problems?

Answer: Naive transmission of the bytes of the array may result in each integer appearing to have had its bytes reversed at the destination end. The standard solution is to agree on an architecture-independent "wire format", with a standard byte order and word length. The sender converts from local format to wire format; the receiver converts from wire format to local format. This solution addresses not only endian-ness, but also word length and alignment issues in more complex data structures. If the sender and receiver happen to share the same machine architecture, they may agree as an optimization to skip the conversions.

5.2 What is the largest positive number in 32-bit two's complement arithmetic? What is the smallest (largest magnitude) negative number? Why are these numbers not the additive inverse of each other?

Answer: The largest is `0x7fffffff` = $2^{31} - 1$ = `2147483647`. The smallest is `0x80000000` = -2^{31} = `-2147483648`. They aren't additive inverses of each other because one bit pattern (`0x00000000`) is reserved for zero.

5.3 (a) Express the decimal number `1234` in hexadecimal.

Answer: `0x4d2`

---

1 Questions © 2015, Morgan Kaufmann Publishers, Inc.; solutions © 2015, Michael L. Scott. This material may not be copied or distributed without written permission of the publisher and author.

(b) Express the unsigned hexadecimal number `0x2ae` in decimal.

Answer: 686.

(c) Interpret the hexadecimal bit pattern `0xffd9` as a 16-bit 2's complement number. What is its decimal value?

Answer: -39.

(d) Suppose that n is a negative integer represented as a k-bit 2's complement bit pattern. If we reinterpret this bit pattern as an unsigned number, what is its numeric value as a function of n and k?

Answer: $n + 2^k$.

5.4 What will the following C code print on a little-endian machine like the x86? What will it print on a big-endian machine?

```
unsigned short n = 0x1234;  // 16 bits
unsigned char *p = (unsigned char *) &n;
printf ("%d\n", *p);
```

Answer: Little-endian: `52`; big-endian: `18`.

5.5 (a) Suppose we have a machine with hardware support for 8-bit integers. What is the decimal value of 11011001_2, interpreted as an unsigned quantity? As a signed, two's complement quantify? What is its two's complement additive inverse?

Answer: Using binary place value, $11011001_2 = 128 + 64 + 16 + 8 + 1 = 217$. Interpreting it as an 8-bit two's complement number, however, we notice that the leading (sign) bit is 1, so the numeric value is $217 - 2^8 = -39$. To negate this value (take its additive inverse), we flip all the bits and add 1: $-11011001_2 = 00100110_2 + 1 = 00100111_2$.

(b) What is the 8-bit binary sum of 11011001_2 and 10010001_2? Does this sum result in overflow if we interpret the addends as unsigned numbers? As signed two's complement numbers?

Answer: Again using binary place value,

$$\begin{array}{r} 11011001_2 \\ 10010001_2 \\ \hline 01101010_2 \end{array}$$

The second addend has a decimal value of 145 if unsigned, or -111 in two's complement. There is a carry out of the high bit of the sum, so overflow occurs when the addends are considered unsigned. Reasoning in decimal, $217 + 145 = 362 > 255 = 2^8 - 1$, the largest representable unsigned value.

If we consider the addends to be unsigned instead, we note that the carries into and out of the most significant bit (0 and 1, respectively) are different, so the sum still overflows, though for a different reason. In decimal, we have $-39 + (-111) = -150 < -128 = -2^7$, the smallest representable signed value.

5.6 In Section C-5.2.1 we observed that overflow occurs in two's complement addition when we add two non-negative numbers and obtain an apparently negative result, or add two negative numbers and obtain an apparently non-negative result. Prove that it is equivalent to say that a two's complement addition operation overflows if and only if the carry into most significant place differs from the carry out of most significant place. (This trivial check is the one typically performed in hardware.)

Answer: The proof is a simple exercise in case analysis. If we add a negative number and a non-negative number, a carry out of the left-most place will happen iff there is a carry into the left-most place ($1 + 0 + 0 = 1$, with no carry; $1 + 0 + 1 = 0$, with a carry). This case never overflows, and the carries in and out are the same. If we add two negative numbers, we will obtain an apparently non-negative result (an overflow) iff there is no carry into the left-most place ($1 + 1 + 0 = 0$, with a carry). If we add two non-negative numbers, we will obtain an apparently negative result (an overflow) iff there *is* a carry into the left-most place ($0 + 0 + 1 = 1$, with no carry). In both these latter cases, overflow occurs iff the carries differ.

5.7 In Section C-5.2.1 we claimed that a two's complement integer could be correctly negated by flipping the bits, adding 1, and discarding any carry out of the left-most place. Prove that this claim is correct.

Answer: The defining characteristic of n-bit two's complement arithmetic is that $k + -k = 0$, where 0 is represented by a string of n zeros. Let $\tilde{k}$ be the number we obtain by flipping all the bits in k. If we add k and $\tilde{k}$ using standard positional addition, we clearly obtain a string of ones, which equals $2^n - 1$, unsigned, where n is the width of a word in bits. If we add an extra one the carries ripple all the way across, producing a carry out of the most significant place, which is discarded in unsigned arithmetic, leaving 0. Thus $(k + \tilde{k}) + 1 = 0$, or $k + (\tilde{k} + 1) = 0$, or $-k = (\tilde{k} + 1)$. All arithmetic is carried out in the ring of integers modulo n, in which addition is associative.

5.8 What is the single-precision IEEE floating-point number whose value is closest to 6.022×10^{23}?

Answer: $6.022 \times 10^{23} = 1.1111\ 1110\ 0001\ 0101\ 0100\ 1111\ 01\ldots_2 \times 2^{78}$, so we want $s = 1$, $e = 78 + 127 = 205 = 1100\ 1101_2$, and $f = 1111\ 1110\ 0001\ 0101\ 0101\ 000$, giving us a bit pattern of `1110 0110 1111 1111 0000 1010 1010 1000`, or `1110 0110 1111 1111 0000 1010 1010 0111` if rounded down.

5.9 Occasionally one sees a C program in which a double-precision floating-point number is used as an integer counter. Why might a programmer choose to do this?

Answer: Because a double-precision floating-point number has 52 bits of significand, which is more than the bits available to the typical integer on a 32-bit machine. If we use both numbers as counters, the integer will overflow long before the floating-point number suffers a loss of precision. On some machines floating-point arithmetic on the double may be faster than software algorithms for extended precision (multi-word) integer arithmetic, though this is far from certain.

5.10 Modern compilers often find they don't have enough registers to hold all the things they'd like to hold. At the same time, VLSI technology has reached the

point at which there is room on a chip to hold many more registers than are found in the typical ISA. Why are we still using instruction sets with only 32 integer registers? Why don't we make, say, 64 or 128 of them visible to the programmer?

Answer: Because pipelining is much easier if instructions are all the same length. If we increase the number of registers then we'll need more bits to encode each register name, and we'll either have to have longer instructions (which makes programs bigger, and increases the need for memory and bus bandwidth to read the instructions in from memory), or we'll have to steal bits from something else, like immediate (within-the-instruction) constants, where we really don't have them to spare.

5.11 Some early RISC machines (SPARC among them) provided a "multiply step" instruction that performed one iteration of the standard shift-and-add algorithm for binary integer multiplication. Speculate as to the rationale for this instruction.

Answer: Multiplication takes longer than addition. In mid 1980s technology there wasn't enough room on a chip to include a separate integer multiplier with its own separate (longer) pipeline. To avoid lengthening the pipeline of the main ALU (thereby delaying all other arithmetic instructions), the designers opted to include an instruction that would do as much of an integer multiply as it could in a single cycle.

5.12 Why do you think RISC machines standardized on 32-bit instructions? Why not some smaller or larger length? With the notable exception of ARM, why not multiple lengths?

Answer: A single length of instruction dramatically simplifies parallel instruction decoding: the hardware can tell where later instructions begin before it has finished decoding earlier instructions. It also makes sense for instructions to be a power of two bytes in length, so instructions are aligned uniformly in memory. Two bytes isn't enough space to specify two source registers, a target register, and an opcode on a machine with more than about 8 registers. Eight bytes is more than we usually need; if it were the standard length we'd waste a *lot* of bits. Four bytes gives us enough space for most purposes. The principal exceptions are very large displacements and immediate operands, which must be handled with two-instruction sequences.

5.13 Consider a machine with three condition codes, N, Z, and O. N indicates whether the most recent arithmetic operation produced a negative result. Z indicates whether it produced a zero result. O indicates whether it produced a result that cannot be represented in the available precision for the numbers being manipulated (i.e., outside the range $0\,..\,2^n$ for unsigned arithmetic, $-2^{n-1}\,..\,2^{n-1}-1$ for signed arithmetic). Suppose we wish to branch on condition `A op B`, where `A` and `B` are unsigned binary numbers, for `op` $\in \{<, \leq, =, \neq, >, \geq\}$. 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.

Answer:

op	signed	unsigned
$<$	$(N \wedge \neg O) \vee (O \wedge \neg N)$	N
$\leq$	$Z \vee (N \wedge \neg O) \vee (O \wedge \neg N)$	$N \vee Z$
$=$	Z	Z
$\neq$	$\neg Z$	$\neg Z$
$>$	$(N \wedge O) \vee (\neg N \wedge \neg O \wedge \neg Z)$	$\neg N \wedge \neg Z$
$\geq$	$(N \wedge O) \vee (\neg N \wedge \neg O)$	$\neg N$

5.14 We implied in Section C-5.4.1 that if one adds a new instruction to a nonpipelined, 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?

Answer: As the instruction set grows, the time required for instruction decode increases, as does the size of the microprogram. At certain thresholds the number of bits in a microinstruction must increase, along with the time required to decode it, in order to accommodate branches. Storage for the microprogram also consumes chip area that could be used for such things as faster adders. If we attempt to condense the microprogram, e.g., by breaking common microinstruction sequences out as microsubroutines, then again it may run slower.

5.15 Suppose that loads constitute 25% of the typical instruction mix on a certain machine. Suppose further that 15% of these loads miss in the last level of on-chip cache, with a penalty of 120 cycles to reach main memory. What is the contribution of last-level cache misses to the average number of cycles per instruction? You may assume that instruction fetches always hit in the L1 cache. Now suppose that we add an off-chip (L3 or L4) cache that can satisfy 90% of the misses from the last-level on-chip cache, at a penalty of only 30 cycles. What is the effect on cycles per instruction?

Answer: $0.25 \times 0.15 \times 120 = 4.5$ cycles.
$0.25 \times 0.15 \times (0.9 \times 30 + 0.1 \times 120) = 0.0375 \times (27 + 12) = 1.4625$ cycles.

5.16 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?).

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.

5.17 With the development of deeper, more complex pipelines, delayed loads and branches became significantly less appealing as features of a RISC instruction set. In later generations, architects eliminated visible load delays but were unable to do so for branches. Explain.

Answer: The problem is the need for backward compatibility. If we eliminate delayed loads from an architecture, old programs will still run correctly; any `nop`s they contain in delay slots will be harmless. (Of course, new programs will not run correctly on old machines, because they lack the `nop`s.) 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.

5.18 Some processors, including the Power series and certain 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?

Answer: Much the same options available for load delays. Ideally, one would schedule an instruction that has no impact on the condition. This is harder on the x86 than the Power series, because so many instructions set the condition codes on the x86. Power processors, by contrast, have multiple sets of condition codes; an instruction that uses a different set can safely be scheduled into the delay. Of course, on an out-of-order processor the compiler can leave the task of finding an appropriate instruction to the run-time instruction scheduler. On the x86, that scheduler may benefit from hardware condition code renaming.

5.19 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?

Answer: Static prediction obviously avoids chip-area and run-time energy costs. It also facilitates instruction scheduling across branches (see the discussion of *trace scheduling* under "Dynamic Optimization" in Section 16.2.2). Run-time prediction, however, can exploit input- or phase-dependent behavior, and is usually much more effective. Fortunately, the two are not mutually exclusive: the compiler can make predictions for the purpose of instruction scheduling and the hardware can use its own predictions to keep the pipeline full.

5.20 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?

Answer: Let C represent cycles per instruction (CPI) with perfect prediction. Originally, CPI $= 0.8C + 0.2 \times (0.75C + 0.25 \times (C + 3)) = C + 0.15$. With better prediction, CPI $= C + 0.06$.
Percent slowdown for C=1.5: 0.15/1.5 = 10% or 0.06/1.5 = 4%.
Percent slowdown for C=0.6: 0.15/0.6 = 25% or 0.06/0.6 = 10%.
Branch prediction increases in importance with the degree of instruction-level parallelism in the processor.

5.21 Consider the code in Figure C-5.9. In an attempt to eliminate the remaining delay, and reduce the overhead of the bookkeeping (loop control) instructions, one might consider *unrolling* the loop: 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 machine supports displacement addressing. Schedule instructions as tightly as you can. How many cycles does your loop consume per vector element?

Answer:

6 Control Flow

6.10 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

6.1 We noted in Section 6.1.1 that most binary arithmetic operators are left-associative in most programming languages. In Section 6.1.4, however, we also noted that most compilers are free to evaluate the operands of a binary operator in either order. Are these statements contradictory? Why or why not?

Answer: No, they are not contradictory. When there are consecutive identical operators within an expression, associativity determines which subexpressions are arguments of which operators. It does not determine the order in which those subexpressions are evaluated. For example, left associativity for subtraction determines that `f(a) - g(b) - h(c)` groups as `(f(a) - g(b)) - h(c)` (rather than `f(a) - (g(b) - h(c))`, but it does not determine whether `f` or `g` is called first.

6.2 As noted in Figure 6.1, Fortran and Pascal give unary and binary minus the same level of precedence. Is this likely to lead to nonintuitive evaluations of certain expressions? Why or why not?

Answer: Probably not in the most common cases. Left-to-right evaluation will cause unary minus to "group" more tightly than other operators at the same level; there is thus no problem with `-A + B`. More significantly, the associativity of multiplication means that results will usually be the same regardless of the order in which negation is applied:

6.3 In Example 6.9 we described a common error in Pascal programs caused by the fact that `and` and `or` have precedence comparable to that of the arithmetic operators. Show how a similar problem can arise in the stream-based I/O of C++ (described in Section C-8.7.3). (Hint: Consider the precedence of << and >>, and the operators that appear below them in the C column of Figure 6.1.)

Answer: If we write

```
cout << 1234 & 5678 << "\n";
```

the compiler is likely to provide an unhelpful message along the following lines:

```
error: invalid operands of types 'int' and 'const char [2]'
to binary 'operator<<'
```

The problem is that precedence groups our code as

```
(cout << 1234) & (5678 << "\n");
```

The error message is complaining about the second "parenthesized" expression. (Thanks to Richard Botting for this exercise.)

6.4 Translate the following expression into postfix and prefix notation:

$$[-b + \mathrm{sqrt}(b \times b - 4 \times a \times c)]/(2 \times a)$$

Do you need a special symbol for unary negation?

Answer: Let ~ represent unary negation (yes, it's needed).
Postfix: `b ~ b b * 4 a * c * - sqrt + 2 a * /`.
Prefix: `/ + ~ b sqrt - * b b * * 4 a c * 2 a`.

6.5 In Lisp, most of the arithmetic operators are defined to take two or more arguments, rather than strictly two. Thus `(* 2 3 4 5)` evaluates to 120, and `(- 16 9 4)` evaluates to 3. Show that parentheses are necessary to disambiguate arithmetic expressions in Lisp (in other words, give an example of an expression whose meaning is unclear when parentheses are removed).

In Section 6.1.1 we claimed that issues of precedence and associativity do not arise with prefix or postfix notation. Reword this claim to make explicit the hidden assumption.

Answer: Without parentheses, would `- 2 3 * 4 5 6` evaluate to `(- 2 3 (* 4 5) 6) = -27` or to `(- 2 3 (* 4 5 6)) = -121`?

More accurately, issues of precedence and associativity do not arise with prefix or postfix notation in which each operator takes a fixed number of operands.

6.6 Example 6.33 claims that "For certain values of `x`, `(0.1 + x) * 10.0` and `1.0 + (x * 10.0)` can differ by as much as 25%, even when `0.1` and `x` are of the same magnitude." Verify this claim. (*Warning:* If you're using an x86 processor, be aware that floating-point calculations [even on single-precision variables] are performed internally with 80 bits of precision. Roundoff errors will appear only when intermediate results are stored out to memory [with limited precision] and read back in again.)

Answer: The single-precision representation of `0.1` is `0x3dcccccd`. For `x`, chose the value obtained by negating that and then flipping the least significant bit of the mantissa: `0x8dcccccc`. The code below outputs

```
a = 3dcccccd 1.000000e-01
b = bdcccccc -9.999999e-02
a + b = 32000000 7.450581e-09
a*10 = 3f800000 1.000000e+00
b*10 = bf7fffff -9.999999e-01
(a+b)*10 = 33a00000 7.450581e-08
(a*10) + (b*10) = 33800000 5.960464e-08
```

The values on the last two lines differ by almost exactly 25%.

The `volatile floats` `s` and `t` serve to force intermediate values through memory on an x86; on most other processors (ARM, SPARC, MIPS, Power, ...) the "commented-out" lines can be used in place of the immediately preceding code blocks.

```
#include <stdio.h>

union {
    volatile float f;
    volatile int i;
} a, b, c, d, e;

volatile float s;
volatile float t;

void test() {

    a.f = 0.1;
    b.i = 0xbdcccccc;

    printf("a = %x %e\n", a.i, a.f);
    printf("b = %x %e\n", b.i, b.f);

    c.f = (float) (a.f + b.f);
    printf("a + b = %x %e\n", c.i, c.f);

    d.f = (float) (a.f * (float) 10.0);
    e.f = (float) (b.f * (float) 10.0);

    printf("a*10 = %x %e\n", d.i, d.f);
    printf("b*10 = %x %e\n", e.i, e.f);

    t = a.f + b.f;
    d.f = t * (float) 10.0;
/*
    d.f = (float) (a.f + b.f) * (float) 10.0;
*/
    printf("(a+b)*10 = %x %e\n", d.i, d.f);
```

```
    s = a.f * (float) 10.0;
    t = b.f * (float) 10.0;
    e.f = s + t;
/*
    e.f = (float) (a.f * (float) 10.0) + (float) (b.f * (float) 10.0);
*/
    printf("(a*10) + (b*10) = %x %e\n\n", e.i, e.f);
}
```

6.7 Is `&(&i)` ever valid in C? Explain.

Answer: No. The `&` operator always takes as argument an expression that has a location. Like most operators, it generates a value that doesn't have a location—one to which the `&` operator therefore cannot be applied. By analogy, variables `a` and `b` may both have locations (so `&a` and `&b` both make sense), but `a + b` does not have a location, so `&(a + b)` does not make sense.

6.8 Languages that employ a reference model of variables also tend to employ automatic garbage collection. Is this more than a coincidence? Explain.

Answer: Yes, it's more than a coincidence. Where a language with a value model of variables tends to propagate copies of objects, a language with a reference model tends to propagate references to the same object. This proliferation of references makes it very difficult for the programmer to keep track of the number of references to any given object, and specifically of whether a given object is no longer referenced at all. Garbage collection becomes more or less essential.

6.9 In Section 6.1.2 ("Orthogonality"), we noted that C uses `=` for assignment and `==` for equality testing. The language designers state: "Since assignment is about twice as frequent as equality testing in typical C programs, it's appropriate that the operator be half as long" [KR88, p. 17]. What do you think of this rationale?

Answer: It's not entirely clear whether this statement was meant to be serious or humorous. Economy of syntax at this level does not appear to be a compelling concern. More relevant, arguably, is the likelihood of confusion: can the wrong operator easily be used by mistake, and can a human reader of the code misinterpret the programmer's intent? The problem in C is not so much the specific choice of operators for assignment and equality testing, but rather the *similarity* of those operators and the fact that both are permissible in an expression context.

6.10 Consider a language implementation in which we wish to catch every use of an uninitialized variable. In Section 6.1.3 we noted that for types in which every possible bit pattern represents a valid value, extra space must be used to hold an initialized/uninitialized flag. Dynamic checks in such a system can be expensive, largely because of the address calculations needed to access the flags. We can reduce the cost in the common case by having the compiler generate code to automatically initialize every variable with a distinguished *sentinel* value. If at some point we find that a variable's value is different from the sentinel, then that variable must have been initialized. If its value *is* the sentinel, we must double-check the flag. Describe a plausible allocation strategy for initialization flags, and show

the assembly language sequences that would be required for dynamic checks, with and without the use of sentinels.

Answer: For dynamically allocated variables, an initialization flag can be included in the header of the variable's heap block, along with the usual size, type, or list pointer metadata. For static and stack-allocated data, the compiler can arrange for per-scope bit arrays in which the *k*th bit indicates the initialization status of the *k*th variable. Zero should probably indicate "uninitialized", because all-zero arrays for global/static data can be placed in zero-fill memory, which need not occupy space in the object file.

Suppose that foo is the *k*th global variable, and that we are running on a *w*-bit machine. Without sentinels, the assembly language to read a global/static variable would look something like this:

```
    r2 := ⌊(k + w − 1)/w⌋ + flag_array_base
        -- address of word containing foo's flag; compile-time constant
    r2 := *r2                       -- load word into register
    r1 := (01 << (k mod w))         -- mask with foo's bit set; compile-time constant
    r2 &:= r1                       -- not 0 if and only if foo is initialized
    if r2 = 0 goto error_handler
    r1 := &foo
    r1 := *r1                       -- r1 contains foo
```

With sentinels, it would look something like this:

```
    r1 := &foo
    r1 := *r1                       -- foo?
    if r1 ≠ sentinel goto ok
    r2 := ⌊(k + w − 1)/w⌋ + flag_array_base
    r2 := *r2
    r3 := (01 << (k mod w))
    r2 &:= r3
    if r2 = 0 goto error_handler
ok: ...                             -- r1 contains foo
```

Access sequences for stack-allocated variables would be similar, but would use displacement addressing with respect to the frame pointer (or appropriate static link) for both the variable and the flag array.

We have assumed here the use of scalar variables. Record fields could be treated, recursively, as if they were separate scalars. Arrays would have a bit per element in the flag array, and would require additional code in the general case to calculate the bit address corresponding to a dynamically calculated index.

Complications arise when passing a variable by reference or creating a pointer to it. Perhaps the simplest approach would be to perform the check at the time the reference is created, and to prohibit the use of references to uninitialized data. Unfortunately, his would seem to preclude programming idioms in which a subroutine is used for initialization. Parameters could be supported by including a hidden initialization flag array among the actual parameters. The called routine would use these flags when accessing the formal parameters, and the values would be copied back on return. Pointers could be extended to include a reference to the initialization flag (address and bit number) in addition to the data reference, but it seems hard to justify the space this would require.

6.11 Write an attribute grammar, based on the following context-free grammar, that accumulates jump code for Boolean expressions (with short-circuiting) into a synthesized attribute code of *condition*, and then uses this attribute to generate code for `if` statements.

stmt ⟶ `if` *condition* `then` *stmt* `else` *stmt*
⟶ *other_stmt*
condition ⟶ *c_term* | *condition* `or` *c_term*
c_term ⟶ *c_factor* | *c_term* `and` *c_factor*
c_factor ⟶ `ident` *relation* `ident` | `(` *condition* `)` | `not` `(` *condition* `)`
relation ⟶ `<` | `<=` | `=` | `<>` | `>` | `>=`

You may assume that the code attribute has already been initialized for *other_stmt* and `ident` nodes. (For hints, see Fischer et al.'s compiler book [FCL10, Sec. 14.1.4].)

Answer: In addition to synthesized attribute code, found on everything other than parentheses, operators, and `not`, we employ three inherited attributes: (1) fall_through_case, found on *condition*, *c_term*, and *c_factor*, indicates whether code should fall through when the experssion is true (and branch when false) or vise versa; (2) label, also found on *condition*, *c_term*, and *c_factor*, gives the location to which to jump in the non-fall-through case. (3) reverse, found on *relation*, indicates whether the sense of the comparison should be reversed in order to maintain consistency with fall_through_case.

As is customary in intermediate code (see Section 15.1.1), we assume an unlimited number of *virtual registers*, allocated by subroutine get_reg. A similar subroutine, get_label, returns a new label name.

stmt$_1$ ⟶ `if` *condition* `then` *stmt*$_2$ `else` *stmt*$_3$
▷ condition.fall_through_case := true
▷ local string L1 := make_new_label() -- beginning of stmt$_3$
▷ local string L2 := make_new_label() -- after stmt$_3$
▷ condition.label := L1
▷ stmt$_1$.code := condition.code
+ stmt$_2$.code + "goto " + L2 + "\n"
+ L1 + ": " + stmt$_3$.code + L2 + ": "

stmt ⟶ *other_stmt*
▷ stmt.code := other_stmt.code

condition ⟶ *c_term*
▷ c_term.fall_through_case := condition.fall_through_case
▷ c_term.label := condition.label
▷ condition.code := c_term.code

condition$_1$ ⟶ *condition*$_2$ `or` *c_term*
▷ local string L := make_new_label() -- after c_term
▷ condition$_2$.fall_through_case := false
-- evaluate c_term only if condition$_2$ is false
▷ condition$_2$.label := if condition$_1$.fall_through_case = false
then condition$_1$.label else L

▷ c_term.fall_through_case := condition$_1$.fall_through_case
▷ c_term.label := condition$_1$.label
▷ condition$_1$.code := condition$_2$.code + c_term.code + L + ": "

c_term ⟶ *c_factor*
▷ c_factor.fall_through_case := c_term.fall_through_case
▷ c_factor.label := c_term.label
▷ c_term.code := c_factor.code

c_term$_1$ ⟶ *c_term*$_2$ `and` *c_factor*
▷ local string L := make_new_label() -- after c_factor
▷ c_term$_2$.fall_through_case := true
  -- evaluate c_factor only if c_term$_2$ is true
▷ c_term$_2$.label := if c_term$_1$.fall_through_case = true
  then c_term$_1$.label else L
▷ c_factor.fall_through_case := c_term$_1$.fall_through_case
▷ c_factor.label := c_term$_1$.label
▷ c_term$_1$.code := c_term$_2$.code + c_factor.code + L + ": "

c_factor ⟶ `ident` *relation* `ident`
▷ relation.reverse := ! c_factor.fall_through_case
▷ local string R1 := get_reg(); local string R2 := get_reg()
▷ c_factor.code := R1 + " := " + ident$_1$.code + "\n"
  + R2 + " := " + ident$_2$.code + "\n"
  + "if " + R1 + relation.code + R2 + " goto " + c_factor.label + "\n"

c_factor ⟶ `(` *condition* `)`
▷ condition.fall_through_case := c_factor.fall_through_case
▷ condition.label := c_factor.label
▷ c_factor.code := condition.code

c_factor ⟶ `not (` *condition* `)`
▷ condition.fall_through_case := ! c_factor.fall_through_case
▷ condition.label := c_factor.label
▷ c_factor.code := condition.code

relation ⟶ `<`
▷ relation.code := if relation.reverse then " ≥ " else " < "

relation ⟶ `<=`
▷ relation.code := if relation.reverse then " > " else " ≤ "

relation ⟶ `=`
▷ relation.code := if relation.reverse then " ≠ " else " = "

relation ⟶ `<>`
▷ relation.code := if relation.reverse then " = " else " ≠ "

relation ⟶ `>`
▷ relation.code := if relation.reverse then " ≤ " else " > "

6.12 Describe a plausible scenario in which a programmer might wish to avoid short-circuit evaluation of a Boolean expression.

Answer: Suppose we wish to count occurrences of words in a document, and print a list of all misspelled words that appear 10 or more times, together with a count of the total number of misspellings. Pascal code for this task might look something like the following:

```
 1.  function tally(word : string) : integer;
 2.      (* Look up word in hash table.  If found, increment tally; If not
 3.          found, enter with a tally of 1.  In either case, return tally. *)
     ...
 4.  function misspelled(word : string) : Boolean;
 5.      (* Check to see if word is mis-spelled and return appropriate
 6.          indication.  If yes, increment global count of mis-spellings. *)
     ...
 7.  while not eof(doc_file) do begin
 8.      w := get_word(doc_file);
 9.      if (tally(w) = 10) and misspelled(w) then
10.          writeln(w)
11.  end;
12.  writeln(total_misspellings);
```

Here the `if` statement at line 9 tests the conjunction of two subexpressions, both of which have important side effects. If short-circuit evaluation is used, the program will not compute the right result. The code can be rewritten to eliminate the need for non-short-circuit evaluation, but one might argue that the result is more awkward than the version shown.

6.13 Neither Algol 60 nor Algol 68 employs short-circuit evaluation for Boolean expressions. In both languages, however, an `if... then ... else` construct can be used as an expression. Show how to use `if... then ... else` to achieve the effect of short-circuit evaluation.

Answer: Ada's `A and then B` can be written `if A then B else false` in Algol 68. Ada's `A or else B` can be written `if A then true else B` in Algol 68.

6.14 Consider the following expression in C: `a/b > 0 && b/a > 0`. What will be the result of evaluating this expression when `a` is zero? What will be the result when `b` is zero? Would it make sense to try to design a language in which this expression is guaranteed to evaluate to `false` when either `a` or `b` (but not both) is zero? Explain your answer.

Answer:

6.15 As noted in Section 6.4.2, languages vary in how they handle the situation in which the controlling expression in a `case` statement does not appear among the labels on the arms. C and Fortran 90 say the statement has no effect. Pascal and Modula say it results in a dynamic semantic error. Ada says that the labels must *cover* all possible values for the type of the expression, so the question of a missing value can never arise at run time. What are the tradeoffs among these alternatives? Which do you prefer? Why?

Answer:

6.16 The equivalence of `for` and `while` loops, mentioned in Example 6.64, is not precise. Give an example in which it breaks down. Hint: think about the `continue` statement.

Answer:

6.17 Write the equivalent of Figure 6.5 in C# or Ruby. Write a second version that performs an in-order enumeration, rather than preorder.

Answer:

6.18 Revise the algorithm of Figure 6.6 so that it performs an in-order enumeration, rather than preorder.

Answer:

6.19 Write a C++ preorder iterator to supply tree nodes to the loop in Example 6.69. You will need to know (or learn) how to use pointers, references, inner classes, and operator overloading in C++. For the sake of (relative) simplicity, you may assume that the data in a tree node is always an `int`; this will save you the need to use generics. You may want to use the `stack` abstraction from the C++ standard library.

Answer:

```
#include <stack>
#include <iostream.h>
using namespace std;

class tree_node {
    tree_node* left;
    tree_node* right;
public:
    int val;
    tree_node(tree_node *l, tree_node *r, int v) {
        left = l; right = r; val = v;
    }
    tree_node(int v) {
        left = 0; right = 0; val = v;
    }

    class iterator {
        stack<tree_node*> *s;
        tree_node *cur;
```

```
    public:
        iterator(tree_node& n) {
            s = new stack<tree_node*>();
            cur = &n;
            if (cur) {
                if (cur->right) s->push(cur->right);
                if (cur->left) s->push(cur->left);
            }
        }
        ~iterator() {
            delete s;
        }
        bool operator!=(void *p) const {
            return (cur != 0);
        }
        iterator& operator++() {
            if (s->empty()) {
                cur = 0;
            } else {
                cur = s->top();
                s->pop();
                if (cur) {
                    if (cur->right) s->push(cur->right);
                    if (cur->left) s->push(cur->left);
                }
            }
            return *this;
        }
        int& operator*() const{
            return cur->val;
        }
        int* operator->() const{
            return &(cur->val);
        }
    };
    iterator& begin() {
        iterator *i = new iterator(*this);
        return *i;
    }
    void *end() {
        return 0;
    }
};
```

6.20 Write code for the `tree_iter` type (`struct`) and the `ti_create`, `ti_done`, `ti_next`, `ti_val`, and `ti_delete` functions employed in Example 6.73.

Answer:

```
typedef struct tree_node {
    data val;
    struct tree_node *left, *right;
} tree_node;

/* a tree_iter contains a list-based stack of
   tree_nodes (subtrees) that have yet to be visited */

typedef struct list_node {
    tree_node *tn;
    struct list_node *next;
} list_node;

typedef list_node *tree_iter;

void ti_push(tree_node *n, tree_iter *tp) {
    if (n) {
        list_node *t = *tp;
        *tp = (tree_iter) malloc(sizeof(list_node));
        (*tp)->tn = n;  (*tp)->next = t;
    }
}

void ti_create(tree_node *root, tree_iter *tp) {
    *tp = 0;
    ti_push(root, tp);
}

int ti_done(tree_iter ti) {return (ti == 0);}

data ti_val(tree_iter ti) {
    assert(ti != 0);
    return ti->tn->val;
}

void ti_next(tree_iter *tp) {
    tree_node *n = (*tp)->tn;
    tree_iter t = (*tp)->next;
    free(*tp);
    *tp = t;
    ti_push(n->right, tp);  ti_push(n->left, tp);
}
```

```
void ti_delete(tree_iter *tp) {
    while (*tp != 0) {
        tree_iter t = (*tp)->next;
        free(*tp);
        *tp = t;
    }
}
```

6.21 Write, in C#, Python, or Ruby, an iterator that yields

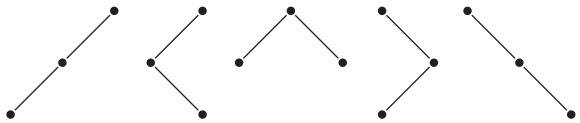
(a) all permutations of the integers $1 \ldots n$

(b) all combinations of k integers from the range $1 \ldots n$ ($0 \leq k \leq n$).

You may represent your permutations and combinations using either a list or an array.

Answer:

6.22 Use iterators to construct a program that outputs (in some order) all *structurally distinct* binary trees of n nodes. Two trees are considered structurally distinct if they have different numbers of nodes or if their left or right subtrees are structurally distinct. There are, for example, five structurally distinct trees of three nodes:



These are most easily output in "dotted parenthesized form":

```
(((.).).)
((.(.)).)
((.).(.))
(.((.).))
(.(.(.)))
```

(Hint: Think recursively! If you need help, see Section 2.2 of the text by Finkel [Fin96].)

Answer: The following is a complete solution using true iterators in Clu. Similar code can be written in Python, Ruby, or C#. The equivalent program with iterator objects (in C++ or Java) is considerably messier.

```
bin_tree = cluster is tree_gen, print
    node = record [left, right: bin_tree]
    rep = variant [some: node, empty: null]
```

```
    tree_gen = iter(k: int) yields(cvt)
        if k = 0 then yield(rep$make_empty(nil))
            % just the empty tree
        else
            for i : int in int$from_to(1, k) do
                for l : bin_tree in tree_gen(i-1) do
                    for r : bin_tree in tree_gen(k-i) do
                        yield(rep$make_some(node${left: l, right: r}))
                    end
                end
            end
        end
    end tree_gen

    print = proc(t: cvt)
        po: stream := stream$primary_output()
        tagcase t
            tag empty:
            tag some(n: node):
                stream$putc(po, '(');
                bin_tree$print(n.left)
                stream$putc(po, '.');
                bin_tree$print(n.right)
                stream$putc(po, ')');
        end
    end print
end bin_tree

start_up = proc()
    pi: stream := stream$primary_input()
    po: stream := stream$primary_output()

    s: string := stream$getl(pi)
    n: int := int$parse(s)
           except when bad_format:
                       stream$putl(po, "Illegal integer")
                       n := 0
                  end
    for t: bin_tree in bin_tree$tree_gen(n) do
        bin_tree$print(t)
        stream$putc(po, '\n');
    end
end start_up
```

6.23 Build true iterators in Java using threads. (This requires knowledge of material in Chapter 13.) Make your solution as clean and as general as possible. In particular, you should provide the standard `Iterator` or `IEnumerable` interface, for use with extended `for` loops, but the programmer should not have to write these.

Instead, he or she should write a class with an `Iterate` method, which should in turn be able to call a `Yield` method, which you should also provide. Evaluate the cost of your solution. How much more expensive is it than standard Java iterator objects?

Answer:

6.24 In an expression-oriented language such as Algol 68 or Lisp, a `while` loop (a `do` loop in Lisp) has a value as an expression. How do you think this value should be determined? (How is it determined in Algol 68 and Lisp?) Is the value a useless artifact of expression orientation, or are there reasonable programs in which it might actually be used? What do you think should happen if the condition on the loop is such that the body is never executed?

Answer:

6.25 Consider a mid-test loop, here written in C, that looks for blank lines in its input:

```
for (;;) {
    line = read_line();
    if (all_blanks(line)) break;
    consume_line(line);
}
```

Show how you might accomplish the same task using a `while` or `do` (`repeat`) loop, if mid-test loops were not available. (Hint: One alternative duplicates part of the code; another introduces a Boolean flag variable.) How do these alternatives compare to the mid-test version?

Answer: We could "rotate" the loop and introduce an extra initial `read_line`:

```
line = read_line();
while (!all_blanks(line)) {
    consume_line(line);
    line = read_line();
}
```

Alternatively, we could place the second half of the loop in an `if` statement that replicates the termination test (assuming that test is idempotent):

```
do {
    line = read_line();
    if (!all_blanks(line)) consume_line(line);
} while !(all_blanks(line));
```

With either a `while` loop or a `repeat` loop we could avoid the duplicated code by introducing an extra flag variable:

```
_Bool done = false;
while (!done) {
    line = read_line();
    if (all_blanks(line)) done = true;
    else consume_line(line);
}

_Bool done = false;
do {
    line = read_line();
    if (all_blanks(line)) done = true;
    else consume_line(line);
} while (!done);
```

These solutions all require either extra variables or extra code. The second doesn't work if `all_blanks` has side effects.

6.26 Rubin [Rub87] used the following example (rewritten here in C) to argue in favor of a `goto` statement:

```
int first_zero_row = -1;        /* none */
int i, j;
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        if (A[i][j]) goto next;
    }
    first_zero_row = i;
    break;
next: ;
}
```

The intent of the code is to find the first all-zero row, if any, of an $n \times n$ matrix. Do you find the example convincing? Is there a good structured alternative in C? In any language?

Answer: Here's a possible solution in C:

```
first_zero_row = -1;
for (i = 0; i < n && first_zero_row == -1; i++) {
    first_zero_row = i;
    for (j = 0; j < n; j++) {
        if (A[i][j]) {
            first_zero_row = -1;
            break;
        }
    }
}
```

If C allowed us to use a `continue` statement to jump to the next iteration of a specific, named loop, we could write the following:

```
first_zero_row = -1;        /* none */
outer: for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        if (A[i][j]) continue outer;            /* not C */
    }
    first_zero_row = i;
    break;
}
```

6.27 Bentley [Ben86, Chap. 4] provides the following informal description of binary search:

> We are to determine whether the sorted array `X[1..N]` contains the element `T`.... Binary search solves the problem by keeping track of a range within the array in which `T` must be if it is anywhere in the array. Initially, the range is the entire array. The range is shrunk by comparing its middle element to `T` and discarding half the range. The process continues until `T` is discovered in the array or until the range in which it must lie is known to be empty.

Write code for binary search in your favorite imperative programming language. What loop construct(s) did you find to be most useful? NB: when he asked more than a hundred professional programmers to solve this problem, Bentley found that only about 10% got it right the first time, without testing.

Answer:

6.28 A *loop invariant* is a condition that is guaranteed to be true at a given point within the body of a loop on every iteration. Loop invariants play a major role in *axiomatic semantics*, a formal reasoning system used to prove properties of programs. In a less formal way, programmers who identify (and write down!) the invariants for their loops are more likely to write correct code. Show the loop invariant(s) for your solution to the preceding exercise. (Hint: You will find the distinction between $<$ and $\leq$ [or between $>$ and $\geq$] to be crucial.)

Answer: Invariants in the following are shown in curly braces:

```
l := 1
h := n+1
{l < h}
while l < h
    {l < h}
    {t, if present, is in range [l..h)}
    m = floor ((l+h)/2)
    {l <= m < h}
    if t = X[m]
        exit
    if t < X[m]
        h := m
        {l <= h}
    else
        l := m+1
```

```
        {l <= h}
{found iff X[m] = t} b
```

6.29 If you have taken a course in automata theory or recursive function theory, explain why `while` loops are strictly more powerful than `for` loops. (If you haven't had such a course, skip this question!) Note that we're referring here to Ada-style `for` loops, not C-style.

Answer:

6.30 Show how to calculate the number of iterations of a general Fortran 90-style `do` loop. Your code should be written in an assembler-like notation, and should be guaranteed to work for all valid bounds and step sizes. Be careful of overflow! (Hint: While the bounds and step size of the loop can be either positive or negative, you can safely use an unsigned integer for the iteration count.)

Answer:

6.31 Write a tail-recursive function in Scheme or ML to compute n factorial ($n! = \prod_{1 \le i \le n} i = 1 \times 2 \times \cdots \times n$). (Hint: You will probably want to define a "helper" function, as discussed in Section 6.6.1.)

Answer:

```
(define fact (lambda (n)
  (letrec ((fact_helper
            (lambda (n s)
              (if (= n 0) s (fact_helper (- n 1) (* s n))))))
    (fact_helper n 1))))
```

6.32 Is it possible to write a tail-recursive version of the classic quicksort algorithm? Why or why not?

Answer: Not without continuation-based surgery that would render it unrecognizable. Quicksort depends fundamentally on making *two* recursive calls and returning the concatenation of their results.

6.33 Give an example in C in which an in-line subroutine may be significantly faster than a functionally equivalent macro. Give another example in which the macro is likely to be faster. (Hint: Think about applicative vs normal-order evaluation of arguments.)

Answer: Consider

```
#define abs_m(x) ((x) >= 0 ? (x) : -(x))
inline double abs_f (double x) {return x >= 0 ? x : -x;}
```

Here the function form is likely to be faster, because it avoids evaluating its argument twice.

Now consider

```
#define select_m(n,a,b) ((n) ? (a) : (b))
inline double select_f (int n, double a, double b) {return n ? a : b;}
```

Here the macro form is likely to be faster if the arguments `a` and `b` are expensive to evaluate, because it will evaluate only one of them.

6.34 Use lazy evaluation (`delay` and `force`) to implement iterator objects in Scheme. More specifically, let an iterator be either the null list or a pair consisting of an element and a promise which when `forced` will return an iterator. Give code for an `uptoby` function that returns an iterator, and a `for-iter` function that accepts as arguments a one-argument function and an iterator. These should allow you to evaluate such expressions as

```
(for-iter (lambda (e) (display e) (newline)) (uptoby 10 50 3))
```

Note that unlike the standard Scheme `for-each`, `for-iter` should not require the existence of a list containing the elements over which to iterate; the intrinsic space required for `(for-iter f (uptoby 1 n 1))` should be only $O(1)$, rather than $O(n)$.

Answer:

```
(define uptoby
  (lambda (low high step)
    (letrec ((iter (lambda (n)
                (cons n (delay
                          (let ((m (+ n step)))
                            (if (> m high) '() (iter m))))))))
      (if (> low high) '() (iter low)))))

(define for-iter
  (lambda (f I)
    (if (null? I) '()
      (begin
        (f (car I))
        (for-iter f (force (cdr I)))))))
```

6.35 (Difficult) Use `call-with-current-continuation` (`call/cc`) to implement the following structured nonlocal control transfers in Scheme. (This requires knowledge of material in Chapter 11.) You will probably want to consult a Scheme manual for documentation not only on `call/cc`, but on `define-syntax` and `dynamic-wind` as well.

(a) Multilevel returns. Model your syntax after the `catch` and `throw` of Common Lisp.

Answer:

```
(define catches '())    ; catch stack initially empty

(define-syntax catch
  (syntax-rules ()
    ((catch tag form ...)
     (let* ((old-catches catches)
            (wrapper (lambda (c)
               (set! catches (cons (cons tag c) old-catches))
               form ...)))     ; real work is done here
       (dynamic-wind
         (lambda ()'())
         (lambda () (call-with-current-continuation wrapper))
         (lambda () (set! catches old-catches)))))))

(define throw (lambda (tag v)
  (let ((pop-catches (assq tag catches)))
    (if pop-catches
        ((cdr pop-catches) v)
      (begin
        (display "no matching catch!")
        (newline)
        ('()))))))     ; fatal error
```

This code could be used as follows:

```
(define index
  (lambda (k l)
    (letrec ((helper (lambda (l i)
                (cond
                  ((null? l) '())
                  ((equal? (car l) k) (throw 'found i))
                  (#t (helper (cdr l) (+ 1 i)))))))
      (helper l 0))))
(catch 'found (index "blind" '("three" "blind" "mice"))) ⟹  1
```

(b) True iterators. In a style reminiscent of Exercise 6.34, let an iterator be a function which when `call/cc`-ed will return either a null list or a pair consisting of an element and an iterator. As in that previous exercise, your implementation should support expressions like

```
(for-iter (lambda (e) (display e) (newline)) (uptoby 10 50 3))
```

Where the implementation of `uptoby` in Exercise 6.34 required the use of `delay` and `force`, however, you should provide an `iterator` macro (a Scheme *special form*) and a `yield` function that allows `uptoby` to look like an ordinary tail-recursive function with an embedded `yield`:

```
(define uptoby
  (iterator (low high step)
    (letrec ((helper (lambda (next)
                (if (> next high) '()
                  (begin                ; else clause
                    (yield next)
                    (helper (+ next step)))))))
      (helper low))))
```

Answer:

```
(define yield '())              ; initial implementation is empty
(define-syntax iterator (syntax-rules ()
  ((iterator args stmt ...)
   (lambda args
     (lambda (cont)
       (let ((old_yield yield))
         (dynamic-wind
          (lambda ()
            (set! yield           ; redefine yield for
                  (lambda (v)     ; dynamically nested code
                    (let ((return-thunk
                           (lambda (p) (cont (cons v p)))))
                      (set! cont (call-with-current-continuation
                                  return-thunk))))))
          (lambda () stmt ...)
          (lambda () (set! yield old_yield)))))))))

(define for-iter (lambda (f iter)
  (let* ((p (call-with-current-continuation iter))
         (more (pair? p)))
    (if more
        (let ((v (car p))
              (c (cdr p)))
          (f v) (for-iter f c))
      '()))))
```

IN MORE DEPTH

6.36 (David Hanson [Han93].) Write a program in Icon that will print the k most common words in its input, one per line, with each preceded by a count of the number of times it appears. If parameter k is not specified on the command line, use 10 by default. You will want to consult the Icon manual (available online [GG96]). In addition to `suspend`, `upto`, and `write`, discussed in this text, you may find it helpful to learn about `integer`, `many`, `pull`, `read`, `sort`, `table`, and `tab`. When fed the Gettysburg Address, your program should print:

```
13  that
9   the
8   we
8   to
8   here
7   a
6   and
5   of
5   nation
5   have
```

Answer: Here is Hanson's solution, with minor updates for Icon version 9.4.2:

```
procedure getword(str)
    local word
    str ? while tab(upto(&letters)) do {
        word := tab(many(&letters))
        suspend word
    }
end

procedure main(args)
    local k, words, line
    k := integer(args[1]) | 10
    words := table(0)
    while line := read() do every words[getword(line)] +:= 1
    words := sort(words, 4)
    every 1 to k do write(left(pull(words), 4), pull(words))
end
```

6.37 Write a `findRE` generator in Icon that mimics the behavior of `find`, but takes as its first parameter a regular expression. Use a string to represent your regular expression, with syntax as in Section 2.1.1. Use empty parentheses to represent ϵ. Give highest precedence to Kleene closure, then concatenation, then alternation. You may assume that we never search for vertical bar, asterisk, or parenthesis characters.

Answer:

6.38 Explain why the following guarded commands in SR are *not* equivalent:

```
if a < b -> c := a              if a < b -> c := a
[] b < c -> c := b              [] b < c -> c := b
[] else -> c := d               [] true -> c := d
fi                              fi
```

Answer: SR makes a nondeterministic choice among true guards, but chooses an `else` guard only if all other guards are false. In the guarded command on the left, the third (`else`) arm will be executed *only* if the guards on the first two arms are false. In the guarded command on the right, the third (`true`) arm may be executed even if either or both of the guards on the first two arms are true.

6.39 The astute reader may have noticed that the final line of the code in Example C-6.95 embodies an arbitrary choice. It could just as easily have said `gcd := b`. Show how to use a guarded command to restore the symmetry of the program.

Answer:

```
if true -> gcd := a
[] true -> gcd := b
fi
```

6.40 Write, in SR or pseudocode, a function that returns

(a) an arbitrary nonzero element of a given array

(b) an arbitrary permutation of a given array

In each case, write your code in such a way that if the implementation of nondeterminism were truly random, all correct answers would be equally likely.

Answer:

7 Type Systems

7.1 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

7.1 Most statically typed languages developed since the 1970s (including Java, C#, and the descendants of Pascal) use some form of name equivalence for types. Is structural equivalence a bad idea? Why or why not?

Answer: This question is obviously a matter of opinion. Name equivalence is a more abstract concept, particularly when defined as in Ada. It allows the programmer to specify exactly which types should be considered compatible and which should be considered incompatible, rather than basing this distinction on whether the types share the same implementation. The advantage of structural equivalence (and the reason it is used in ML family languages) is that makes compatibility an *intrinsic* property of types that can be evaluated independent of the contexts in which those types are declared. Among other things, this fact simplifies the implementation of type checking for separately compiled modules; no mechanism is needed to ensure that files are compiled using the same shared type declarations.

7.2 In the following code, which of the variables will a compiler consider to have compatible types under structural equivalence? Under strict name equivalence? Under loose name equivalence?

```
type  T = array [1..10] of integer
      S = T
A  : T
B  : T
C  : S
D  : array [1..10] of integer
```

Answer: All four arrays are structurally equivalent. Under name equivalence, array D is incompatible with the others. Under strict name equivalence A and B are also incompatible with C; under loose name equivalence A, B, and C are all mutually compatible.

7.3 Consider the following declarations:

```
1. type cell              -- a forward declaration
2. type cell_ptr = pointer to cell
3. x : cell
4. type cell = record
5.       val : integer
6.       next : cell_ptr
7. y : cell
```

Should the declaration at line 4 be said to introduce an alias type? Under strict name equivalence, should x and y have the same type? Explain.

Answer: No, it's not an alias. Line 1 is a declaration that isn't a definition: the definition occurs at line 4. Variables x and y share the same type definition, and thus have the same type.

7.4 Suppose you are implementing an Ada compiler, and must support arithmetic on 32-bit fixed-point binary numbers with a programmer-specified number of fractional bits. Describe the code you would need to generate to add, subtract, multiply, or divide two fixed-point numbers. You should assume that the hardware provides arithmetic instructions only for integers and IEEE floating-point. You may assume that the integer instructions preserve full precision; in particular, integer multiplication produces a 64-bit result. Your description should be general enough to deal with operands and results that have different numbers of fractional bits.

Answer: Suppose we wish to compute $c = a\ op\ b$. Suppose further that f_a, f_b, and f_c are the scaling factors for a, b, and c, respectively. That is, for any x, the "natural" 2's complement value of the bits used to represent x must be multiplied by 2^{f_x} to obtain the correct values of x. If $f_a = -3$, for example, then a has three bits to the right of the "binary point." If $f_a = 3$, then a has three implicit zeros to the left of the binary point.

First consider addition and subtraction. If $f = f_a = f_b = f_c$, we can safely use the native addition and subtraction instructions: there is never any loss of precision, and the usual checks for overflow apply. If $f_a \neq f_b$ then one or the other (or both) must be shifted to align the corresponding bits. Whether this will result in overflow or loss of precision depends on the values of a and b, which we won't in general know until run time. We can generate code to perform appropriate checks. Alternatively, assuming that $-1022 \leq f_a, f_b, f_c \leq 1023$, we can convert a and b to double-precision floating-point, compute the (floating-point equivalent of) the result, and convert back to fixed point. Overflow of the fixed-point quantity can be detected simply by examining the floating-point result: double-precision has 52 bits of significand, scaling is explicit, and floating-point overflow results in a NaN.

Multiplication is a little harder. We begin with the native multiplication instruction, producing a 64 bit result scaled by $f_d = f_a + f_b$. If $f_d = f_c$ we are done. Otherwise we perform an arithmetic shift by $f_c - f_d$ bits. If this pushes any nonzero bits off the right end, then we have lost precision.

If the result does not fit in 32 bits (i.e., the upper bits are not all zero or all one), then we have overflow. Both conditions must in general be checked for at run time.

Finally consider division. If we were to perform ordinary integer division on the representations of a and b (i.e. $(a \times 2^{-f_a}) \div (b \times 2^{-f_b})$), the quotient d would be scaled by $f_d = f_b - f_a$. If $f_d = f_c$, this is what we want. If $f_d > f_c$, we can right-shift d by $f_d - f_c$ to get what we want. If $f_d < f_c$, however, d has insufficient precision. In this case we need to shift a by $f_c - f_d$ or b by $f_d - f_c$ (or some combination of the two) prior to performing the division. If a and b cannot be so shifted without losing bits (this must be checked at run time), then c will overflow. An alternative, as with addition and subtraction, is to convert to double-precision floating-point, perform the corresponding floating-point arithmetic operation, and convert the result back to fixed point.

7.5 When Sun Microsystems ported Berkeley Unix from the Digital VAX to the Motorola 680x0 in the early 1980s, many C programs stopped working, and had to be repaired. In effect, the 680x0 revealed certain classes of program bugs that one could "get away with" on the VAX. One of these classes of bugs occurred in programs that use more than one size of integer (e.g., `short` and `long`), and arose from the fact that the VAX is a little-endian machine, while the 680x0 is big-endian (Section C-5.2). Another class of bugs occurred in programs that manipulate both null and empty strings. It arose from the fact that location zero in a Unix process's address space on the VAX always contained a zero, while the same location on the 680x0 is not in the address space, and will generate a protection error if used. For both of these classes of bugs, give examples of program fragments that would work on a VAX but not on a 680x0.

Answer: The following code illustrates an endian-ness bug:

```
void make_uppercase(char *c) {
    if ('a' <= *c && *c <= 'z') {
        *c += 'A'-'a';
    }
}

int c = getchar();
    // getchar is defined to return an int
    // so that end-of-file can be represented by -1
...
make_uppercase(&c);
```

On a VAX the address of the integer `c` is the same as the address of its low-order byte, so function `make_uppercase`, which expects a pointer to a character, will see the right data. On a 680x0, the address of the integer is the address of its high-order byte, and function `make_uppercase` will not look at the right data. (C compilers of the early 1980s did little if any type checking of function parameters.)

The following code illustrates a null string bug:

```
int contains(char *s, char c) {
    // return boolean value indicating whether character c occurs
    // in string s
    while (*s) {
        if (c == *s) return 1;  // true
    }
    return 0;                   // false
}
...
char *get_str() {
    // return pointer to newly-allocated string,
    // or null pointer if there are no more
    ...
}
...
do {
    // fetch lines until we find one that doesn't contain an 'x'
} while (contains(get_str(), 'x');
```

Consider what happens if every line contains an 'x'. At the end of the input, `get_str` will return a null pointer. On a VAX, function `contains` will access the NULL byte at address zero and return false, whereupon the main `do` loop will terminate. On a 680x0, `contains` will take a protection error.

7.6 Ada provides two "remainder" operators, `rem` and `mod` for integer types, defined as follows [Ame83, Sec. 4.5.5]:

> Integer division and remainder are defined by the relation `A = (A/B)*B + (A rem B)`, where `(A rem B)` has the sign of `A` and an absolute value less than the absolute value of `B`. Integer division satisfies the identity `(-A)/B = -(A/B) = A/(-B)`.
>
> The result of the modulus operation is such that `(A mod B)` has the sign of `B` and an absolute value less than the absolute value of `B`; in addition, for some integer value `N`, this result must satisfy the relation `A = B*N + (A mod B)`.

Give values of `A` and `B` for which `A rem B` and `A mod B` differ. For what purposes would one operation be more useful than the other? Does it make sense to provide both, or is it overkill?

Consider also the `%` operator of C and the `mod` operator of Pascal. The designers of these languages could have picked semantics resembling those of either Ada's `rem` or its `mod`. Which did they pick? Do you think they made the right choice?

Answer: If `A` = 5 and `B` = −3, then `A rem B` = 2 and `A mod B` = −1. If `A` = −5 and `B` = 3, then `A rem B` = −2 and `A mod B` = 1.

The `mod` operator facilitates calculation on rings of small integers. It can be used, for example, to index into a circular array. If `M[i]` is an element of an `N`-element circular array, then `M[(i + 1) mod N]` is the next element and `M[(i - 1) mod N]` is the previous element, regardless of where `i` lies within the declared bounds of the array.

As suggested by the wording from the Ada manual, `rem` is useful in conjunction with integer division. Suppose, for example, that we wish to express an `N`-dollar financial transaction in

terms of bills of various denominations, where positive numbers indicate a payment from me to you, and negative numbers indicate a payment from you to me. We might say

```
twenties := N / 20;
N := N rem 20;
tens := N / 10;
N := N rem 10;
fives := N / 5;
ones := N rem 5;
```

Given either `mod` or `rem` one can implement the other, so in some sense having both is overkill. It can, however, be convenient.

C89 semantics for `%` were implementation-dependent. C99 requires the semantics of Ada's `rem`. Pascal requires `b` to be positive; subject to this restriction it provides the semantics of Ada's `mod`.

Which is preferable is clearly a matter of some controversy. One might argue that circular indexing is more common than remaindering, which would argue for `mod` semantics.

7.7 Consider the problem of performing range checks on set expressions in Pascal. Given that a set may contain many elements, some of which may be known at compile time, describe the information that a compiler might maintain in order to track both the elements known to belong to the set and the possible range of unknown elements. Then explain how to update this information for the following set operations: union, intersection, and difference. The goal is to determine (1) when subrange checks can be eliminated at run time and (2) when subrange errors can be reported at compile time. Bear in mind that the compiler *cannot* do a perfect job: some unnecessary run-time checks will inevitably be performed, and some operations that must always result in errors will not be caught at compile time. The goal is to do as good a job as possible at reasonable cost.

Answer: Let us limit our discussion to sets with elements drawn from a modest universe, so bit vectors are a reasonable implementation. One attractive possibility for subrange checking is for the compiler to maintain, for every set expression

- the bounds (L, U) of the universe set
- a bit vector K indicating the elements known to be in the set
- a bit vector M indicating the elements that *might* be in the set

The L and U bounds determine the size and interpretation of the bit vector representation.

For a constant set we know all elements; K and M will be the same. For a set containing an unknown element e, M must include all possible values of e (these may be non-trivial if e has a subrange type). In all cases M will be a superset of K.

For a union operation `A = B + C`,

$$\begin{aligned} L_A &= \min(L_B, L_C) \\ U_A &= \max(U_B, U_C) \\ K_A &= K_B + K_C \\ M_A &= M_B + M_C) \end{aligned}$$

For an intersection operation `A = B * C`,

$$\begin{aligned} L_A &= \max(L_B, L_C) \\ U_A &= \min(U_B, U_C) \\ K_A &= K_B \times K_C \\ M_A &= M_B \times M_C \end{aligned}$$

For a difference operation `A = B - C`,

$$\begin{aligned} L_A &= L_B \\ U_A &= U_B \\ K_A &= K_B - M_C \\ M_A &= M_B - K_C \end{aligned}$$

Now, when a set expression `E` is assigned into a set variable `S`, we must generate run-time checks if M_E contains any elements not in the universe of `S`. We can generate a compile-time error if K_E contains any elements not in the universe of `S`.

7.8 In Section 7.2.2 we introduced the notion of a *universal reference type* (`void*` in C) that refers to an object of unknown type. Using such references, implement a "poor man's generic queue" in C, as suggested in Section 7.3.1. Where do you need type casts? Why? Give an example of a use of the queue that will fail catastrophically at run time, due to the lack of type checking.

Answer:

7.9 Rewrite the code of Figure 7.3 in Ada, Java, or C#.

7.10 (a) Give a generic solution to Exercise 6.19.

(b) Translate this solution into Ada, Java, or C#.

7.11 In your favorite language with generics, write code for simple versions of the following abstractions:

(a) a stack, implemented as a linked list

(b) a priority queue, implemented as a skip list or a partially ordered tree embedded in an array

(c) a dictionary (mapping), implemented as a hash table

Answer:

7.12 Figure 7.3 passes integer `max_items` to the `queue` abstraction as a generic parameter. Write an alternative version of the code that makes `max_items` a parameter to the `queue` constructor instead. What is the advantage of the generic parameter version?

Answer: NB: Like Figure 7.3, this code does not check for overflow or underflow.

```
template<typename item>
class queue {
    item* items;
    int next_free;
    int next_full;
    int max_items;
public:
    queue(int size) {
        next_free = next_full = 0;
        max_items = size;
        items = new item[max_items];
    }
    ~queue() {
        delete items;
    }
    void enqueue(item it) {
        items[next_free] = it;
        next_free = (next_free + 1) % max_items;
    }
    item dequeue() {
        item rtn = items[next_full];
        next_full = (next_full + 1) % max_items;
        return rtn;
    }
};
```

The advantage of the version in Figure 7.3, which passes `max_items` as a generic parameter, is that field `items` can be expanded in line; it doesn't have to be dynamically allocated. We've added a `~queue` destructor to the version here to deallocate `*items` when the queue itself is destroyed. We assume (as does the code in Figure 7.3) that any items remaining in the queue are separately destroyed, if appropriate.

7.13 Rewrite the generic sorting routine of Examples 7.50–7.52 (with constraints) using OCaml or SML functors.

Answer:

7.14 Flesh out the C++ sorting routine of Example 7.53. Demonstrate that this routine does "the wrong thing" when asked to sort an array of `char*` strings.

Answer:

```
#include <iostream>
```

```
template<typename T>
void sort (T A[], int A_size) {
    for (int i = 1; i < A_size; i++) {
        int j = i;  T t = A[i];
        for (; j > 0; j--) {
            if (t >= A[j-1]) break;
            A[j] = A[j-1];
        }
        A[j] = t;
    }
}

int ints[20] =
    {19, 17, 15, 13, 11, 9, 7, 5, 3, 1,
     18, 16, 14, 12, 10, 8, 6, 4, 2, 0};

char *strings[16] =
    {"now", "is", "the", "time", "for", "all", "good", "folk",
    "to", "come", "to", "the", "aid", "of", "the", "party"};

main () {
    for (int i = 0; i < 20; i++)  printf("%d ", ints[i]);
    printf("\n");
    sort (ints, 20);
    for (int i = 0; i < 20; i++)  printf("%d ", ints[i]);
    printf("\n");

    for (int i = 0; i < 16; i++)  printf("%s ", strings[i]);
    printf("\n");
    sort (strings, 16);
    for (int i = 0; i < 16; i++)  printf("%s ", strings[i]);
    printf("\n");
}
```

This program prints:

```
19 17 15 13 11 9 7 5 3 1 18 16 14 12 10 8 6 4 2 0
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
now is the time for all good folk to come to the aid of the party
now is the the the time for all good folk to to come aid of party
```

The C++ compiler is smart enough to create only one static occurrence of the string `"the"`. Three elements of array `strings` point to this same word. Our sort routine compares by address, not content, and the three elements with the same address come out next to each other in the sorted array.

7.15 In Example 7.53 we mentioned three ways to make the need for comparisons more explicit when defining a generic sort routine in C++: make the comparison routine a method of the generic parameter class `T`, an extra argument to the sort routine, or an extra generic parameter. Implement these options and discuss their comparative strengths and weaknesses.

Answer:

7.16 Yet another solution to the problem of the previous exercise is to make the sorting routine a method of a `sorter` class. The comparison routine can then be passed into the class as a constructor argument. Implement this option and compare it to those of the previous exercise.

Answer:

7.17 Consider the following code skeleton in C++:

```
#include <list>
using std::list;

class foo { ...
class bar : public foo { ...

static void print_all(list<foo*> &L) { ...

list<foo*> LF;
list<bar*> LB;
...
print_all(LF);        // works fine
print_all(LB);        // static semantic error
```

Explain why the compiler won't allow the second call. Give an example of bad things that could happen if it did.

Answer: Since every `bar` is a `foo`, it is tempting to think that anything `print_all` might do to elements of a `list<foo*>` (e.g., print them), it should be able to do to elements of a `list<bar*>`, too. But suppose `print_all` inserts a new `foo` object into `L`. After the call returns, the main program might remove an element from `LB` and obtain erroneous behavior, because it expects that object to be of type `bar`. Types `collection<T1>` and `collection<T2>` are incomparable, regardless of any possible inheritance relationship between `T1` and `T2`.

7.18 Bjarne Stroustrup, the original designer of C++, once described templates as "a clever kind of macro that obeys the scope, naming, and type rules of C++" [Str97, 2nd ed., p. 257]. How close is the similarity? What can templates do that macros can't? What do macros do that templates don't?

Answer: The difference between macros and generics is much like the difference between macros and in-line subroutines (Sections 3.7 and 9.2.4): generics are integrated into the rest of the language, and are understood by the compiler, rather than being tacked on as an afterthought, to be expanded by a preprocessor. Generic parameters are type checked. Arguments to generic

subroutines are evaluated exactly once. Names declared inside generic code obey the normal scoping rules. In Ada, which allows nested subroutines and modules, names passed as generic arguments are resolved in the referencing environment in which the instance of the generic was created, but all other names in the generic are resolved in the environment in which the generic itself was declared.

Macros can lead to variable capture and multiple evaluation bugs. Because they are expanded before syntactic analysis, they can lead to very confusing error messages (not that the messages for C++ templates are straightforward!). At the same time, macros sometimes provide useful mechanisms that don't fit in the language's regular syntactic and semantic framework. Examples include access to the current file name and line number, or the ability to join strings together into tokens, which will then be seen as indivisible units by the scanner.

7.19 In Section 9.3.1 we noted that Ada 83 does not permit subroutines to be passed as parameters, but that some of the same effect can be achieved with generics. Suppose we want to apply a function to every member of an array. We might write the following in Ada 83:

```
generic
    type item is private;
    type item_array is array (integer range <>) of item;
    with function F(it : in item) return item;
procedure apply_to_array(A : in out item_array);

procedure apply_to_array(A : in out item_array) is
begin
    for i in A'first..A'last loop
        A(i) := F(A(i));
    end loop;
end apply_to_array;
```

Given an array of integers, `scores`, and a function on integers, `foo`, we can write:

```
procedure apply_to_ints is
    new apply_to_array(integer, int_array, foo);
...
apply_to_ints(scores);
```

How general is this mechanism? What are its limitations? Is it a reasonable substitute for formal (i.e., second-class, as opposed to third-class) subroutines?

Answer: The limitation of this programming idiom is that the generic subroutine `apply_to_array` must be instantiated *after* one has identified the function `F` (here `foo`) that is to be applied to elements of the array. Since subroutines cannot be passed as parameters or stored in variables in Ada, there is a static set of options for `F`, and the compiler can prepare for them all. An Ada program cannot choose a function based on some run-time calculation and then use this function as a generic parameter. Put another way, the Ada program can choose, dynamically, among preexisting instances of `apply_to_array`, but it cannot choose, dynamically, which `F` to use in a given instantiation. There is no notation in which to express the concept of a dynamically chosen function.

7.20 Modify the code of Figure 7.3 or your solution to Exercise 7.12 to throw an exception if an attempt is made to enqueue an item in a full queue, or to dequeue an item from an empty queue.

Answer:

```
class queue_overflow{};
class queue_underflow{};

template<typename item, int max_items>
class queue {
    item items[max_items];
    int next_free;
    int next_full;
    int num_items;
public:
    queue() {
        next_free = next_full = num_items = 0;
    }
    void enqueue(item it) {
        if (num_items == max_items) throw queue_overflow();
        ++num_items;
        items[next_free] = it;
        next_free = (next_free + 1) % max_items;
    }
    item dequeue() {
        if (num_items == 0) throw queue_underflow();
        --num_items;
        item rtn = items[next_full];
        next_full = (next_full + 1) % max_items;
        return rtn;
    }
};
```

IN MORE DEPTH

7.21 C++ has no direct analogue of the `extends X` and `super X` clauses of Java. Why not?

7.22 Write a simple abstract `ordered_set<T>` class (an *interface*) whose methods include `void insert(T val)`, `void remove (T val)`, `bool lookup (T val)`, and `bool is_empty()`, together with a language-appropriate iterator, as described in Section 6.5.3. Using this abstract class as a base, build a simple `list_set` class that uses a sorted linked list internally. Try this exercise in C++, Java, and C#. Note that (in Java and C#, at least), you will need constraints on `T`. Discuss the differences among your implementations.

7.23 Building on the previous exercise, implement higher-level `union<T>`, `intersection<T>`, and `difference<T>` functions that operate on ordered sets. Note that these

should not be members of the `ordered_set<T>` class, but rather stand-alone functions: they should be independent of the details of `list_set` or any other particular `ordered_set`. So, for example, `union(A, B, C)` should verify that `A` is empty, and then add to it all the elements found in `B` or `C`. Explain, for each of C++, Java, and C#, how to handle the comparison of elements.

7.24 Continuing Example C-7.59, the call

```
csNames.consider(null);
```

will generate a run-time exception, because `String.compareTo` is not designed to take `null` arguments.

(a) Modify Figure C-7.6 to guard against this possibility by including a predicate `public Boolean valid(T a);` in the `Chooser<T>` interface, and by modifying `consider` to make an appropriate call to this predicate. Modify class `CaseSensitive` accordingly.

Answer:

```
interface Chooser<T> {
    public Boolean better(T a, T b);
    public Boolean valid(T a);
}

class Arbiter<T> {
    T bestSoFar;
    Chooser<? super T> comp;

    public Arbiter(Chooser<? super T> c) {
        comp = c;
    }
    public void consider(T t) {
        if (comp.valid(t) &&
                (bestSoFar == null || comp.better(t, bestSoFar)))
            bestSoFar = t;
    }
    public T best() {
        return bestSoFar;
    }
}
...
static class CaseSensitive implements Chooser<String> {
    public Boolean better(String a, String b) {
        return a.compareTo(b) < 1;
    }
    public Boolean valid(String a) { return a != null; }
}
```

(b) Suggest how to make similar modifications to the C# `Arbiter` of Figure C-7.8 and Example C-7.66. How should you handle lower bounds when you need both `Better` and `Valid`?

Answer: The two obvious choices, neither of which is entirely satisfactory, are to pass two separate delegates to the `Arbiter` constructor, or to abandon lower bounds.

7.25 (a) Modify your solution to Exercise 7.14 so that the comparison routine is an explicit generic parameter, reminiscent of the `chooser` of Figure C-7.5.

(b) Give an alternative solution in which the comparison routine is an extra parameter to `sort`.

Answer:

7.26 Consider the C++ program shown in Figure C-7.9. Explain why the final call to `first_n` generates a compile-time error, but the call to `last_n` does not. (Note that `first_n` is generic but `last_n` is not.) Show how to modify the final call to `first_n` so that the compiler will accept it.

Answer: Because `first_n` is generic, its arguments must be used to infer the identity of type parameter `T`. To keep the rules from getting completely out of hand, C++ refuses to coerce an argument to make it match a generic parameter. In the call to `last_n`, by contrast, no generic type inference is required, and the compiler is happy to employ the user-supplied coercion from `int_list_box` to `list<int>`. To make the last call to `first_n` acceptable, we must invoke the type conversion explicitly:

```
first_n((list<int>)b, i);
```

7.27 Consider the following code in C++:

```
template <typename T>
class cloneable_list : public list<T> {
public:
    cloneable_list<T>* clone() {
        auto rtn = new cloneable_list<T>();
        for (auto e : *this) {
            rtn->push_back(e);
        }
        return rtn;
    }
};

...
cloneable_list<foo> L;
...
cloneable_list<foo>* Lp = L.clone();
```

Here `*Lp` will be a "deep copy" of `L`, containing a copy of each `foo` object. Try to write equivalent code in Java. What goes wrong? How might you get around the problem?

```
#include <iostream>
#include <list>
using std::cout;
using std::list;

template<typename T> void first_n(list<T> p, int n) {
    for (typename list<T>::iterator li = p.begin(); li != p.end(); li++) {
        if (n-- <= 0) break;
        cout << *li << " ";
    }
    cout << "\n";
}

void last_n(list<int> p, int n) {
    for (list<int>::reverse_iterator li = p.rbegin(); li != p.rend(); li++) {
        if (n-- <= 0) break;
        cout << *li << " ";
    }
    cout << "\n";
}

class int_list_box {
    list<int> content;
public:
    int_list_box(list<int> l) { content = l; }
    operator list<int>() { return content; }
        // user-supplied operator for coercion/conversion
};

int main() {
    int i = 5;
    list<int> l;

    for (int i = 0; i < 10; i++) l.push_back(i);
    int_list_box b(l);

    first_n(l, i);      // works
    last_n(b, i);       // works (coerces b)
    first_n(b, i);      // static semantic error
}
```

Figure 7.5 Coercion and generics in C++. The compiler refuses to accept the final call to `first_n`.

8 Composite Types

8.10 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

8.1 Suppose we are compiling for a machine with 1-byte characters, 2-byte shorts, 4-byte integers, and 8-byte reals, and with alignment rules that require the address of every primitive data element to be an even multiple of the element's size. Suppose further that the compiler is not permitted to reorder fields. How much space will be consumed by the following array? Explain.

```
A : array [0..9] of record
        s : short
        c : char
        t : short
        d : char
        r : real
        i : integer
```

Answer: 240 bytes. Within each element of the array, `s` has offset 0, `c` has offset 2, `t` has offset 4 (forced up 1 by alignment), `d` has offset 6, `r` has offset 8 (forced up 1 by alignment), and `i` has offset 16. Adding the length of `i`, that brings us to 20 bytes, but 20 is not an even multiple of 8. We therefore pad each element out to 24 bytes so that field `r` will be aligned in each array element. `A` has 10 elements, so the total length is 240 bytes.

8.2 In Example 8.10 we suggested the possibility of sorting record fields by their alignment requirement, to minimize holes. In the example, we sorted smallest-alignment-first. What would happen if we sorted longest-alignment-first? Do you see any advantages to this scheme? Any disadvantages? If the record as a

whole must be an even multiple of the longest alignment, do the two approaches ever differ in total space required?

Answer: On most machines, alignment requirements are powers of two, and any field that can be aligned at a a multiple of 2^i can also be aligned at a multiple of 2^{i+1}. On such a machine, sorting longest-alignment-first moves all holes to the end of the record (assuming, of course, that the length of every field is a multiple of its alignment constraint). With no holes in the middle, we can now compare records with a simple loop.

On the negative side, the average offset of fields from the beginning of the record will increase with longest-first sorting. This is probably not a problem in most cases, but for some records on some machines it may reduce the number of fields that can be reached from the beginning of the record using displacement addressing.

If all alignment constraints are powers of two, then the maximum space lost to holes with shortest-first sorting is $1 + 2 + ... + 2^{k-1} < 2^k$. Since the record must presumably be a multiple of 2^k in length, we won't manage to shorten it by moving the holes to the end.

8.3 Give Ada code to map from lowercase to uppercase letters, using

(a) an array

(b) a function

Note the similarity of syntax: in both cases `upper('a')` is `'A'`.

Answer:

(a)
```
subtype lc_letter is character range 'a'..'z';
upper : array (lc_letter) of character :=
    ('A', 'B', 'C', 'D', 'E', 'F',
     'G', 'H', 'I', 'J', 'K', 'L',
     'M', 'N', 'O', 'P', 'Q', 'R',
     'S', 'T', 'U', 'V', 'W', 'X',
     'Y', 'Z');
```

(b)
```
subtype lc_letter is character range 'a'..'z';
function upper(l : lc_letter) return character is
    uc_offset : constant integer :=
        character'pos('A') - character'pos('a');
begin
    return character'val(character'pos(l) + uc_offset);
end upper;
```

8.4 In Section 8.2.2 we noted that in a language with dynamic arrays and a value model of variables, records could have fields whose size is not known at compile time. To accommodate these, we suggested using a dope vector for the record, to track the offsets of the fields.

Suppose instead that we want to maintain a static offset for each field. Can we devise an alternative strategy inspired by the stack frame layout of Figure 8.7, and divide each record into a fixed-size part and a variable-size part? What problems would we need to address? (Hint: Consider nested records.)

Answer: The fixed- and variable-part strategy for stack frames depends on the fact that the stack frame itself does not have to be of static size. If we want record fields to have static offsets, we cannot embed a variable-size record inside another record, even if the inner record is divided into two contiguous pieces. We can make the strategy work, but only if we allow records to be noncontiguous. (We could, for example, use heap allocation for every array whose size is not statically known.) This choice would complicate assignment of records (we could no longer use a simple block-move loop), and it would require that we generate code in each procedure prologue to deallocate any local arrays that were allocated in the heap. (This is, of course, exactly the complication that motivated many language designers not to allow the size of an array to change after elaboration/allocation.) Dope vectors for records are probably a better solution.

8.5 Explain how to extend Figure 8.7 to accommodate subroutine arguments that are passed by value, but whose shape is not known until the subroutine is called at run time.

Answer:

8.6 Explain how to obtain the effect of Fortran 90's `allocate` statement for one-dimensional arrays using pointers in C. You will probably find that your solution does not generalize to multidimensional arrays. Why not? If you are familiar with C++, show how to use its `class` facilities to solve the problem.

Answer:

8.7 Example 8.24, which considered the layout of a two-dimensional array of characters, counted only the space devoted to characters and pointers. This is appropriate if the space is allocated statically, as a global array of days or keywords known at compile time. Supposed instead that space is allocated in the heap, with with 4 or 8 bytes of overhead for each contiguous block of storage. How does this change the tradeoffs in space efficiency?

Answer:

8.8 Consider the array indexing calculation of Example 8.25. Suppose that i, j, and k are already loaded into registers, and that A's elements are integers, allocated contiguously in memory on a 32-bit machine. Show, in the pseudo-assembly notation of Sidebar 5.1, the instruction sequence to load A[i, j, k] into a register. You may assume the existence of an indexed addressing mode capable of scaling by small powers of two. Assuming the final memory load is a cache hit, how many cycles is your code likely to require on a modern processor?

Answer:

```
   -- assume i is in r1, j is in r2, and k is in r3
1. r4 := r1 × S1
2. r5 := r2 × S2
3. r6 := &A − L1 × S1 − L2 × S2 − L3 × 4    -- one or two instructions
4. r6 := r6 + r4
5. r6 := r6 + r5
6. r7 := *r6[r3]                            -- load
```

The multiplications on lines 1 and 2 are likely to result in several delay slots each (Section C-5.5.1). Depending on the number of slots, and on the number of instructions required at line 3, the delay may or may not be hidden by subsequent instructions (in context, the compiler may be able to find additional instructions to move into the delays). The final instruction will also incur a load delay. Assuming we hit in the cache, the total cost of the access will be something on the order of seven to ten cycles. Also, the initial multiplications might best be replaced (e.g., by a peephole optimizer) with combinations of adds, shifts, and subtracts.

8.9 Continuing the previous exercise, suppose that A has row-pointer layout, and that i, j, and k are again available in registers. Show pseudo-assembler code to load A[i, j, k] into a register. Assuming that all memory loads are cache hits, how many cycles is your code likely to require on a modern processor?

Answer:

```
    -- assume i is in r1, j is in r2, and k is in r3
1.  r4 := &A                -- one or two instructions
2.  r4 := *r4[r1]
3.  r4 := *r4[r2]
4.  r7 := r4[r3]
```

Assuming that the loads at lines 2 and 3 hit in the cache, this code will be comparable in cost to the instruction sequence for contiguous allocation in the previous exercise. (If the loads are misses, it will be substantially slower.)

8.10 Repeat the preceding two exercises, modifying your code to include run-time checking of array subscript bounds.

Answer:

8.11 In Section 8.2.3 we discussed how to differentiate between the constant and variable portions of an array reference, in order to efficiently access the subparts of array and record objects. An alternative approach is to generate naive code and count on the compiler's code improver to find the constant portions, group them together, and calculate them at compile time. Discuss the advantages and disadvantages of each approach.

Answer:

8.12 Consider the following C declaration, compiled on a 64-bit x86 machine:

```
struct {
    int n;
    char c;
} A[10][10];
```

If the address of `A[0][0]` is 1000 (decimal), what is the address of `A[3][7]`?

Answer: $1000 + (3 \times 10 \times 16) + (7 \times 16) = 1592$.

8.13 Suppose we are generating code for an imperative language on a machine with 8-byte floating-point numbers, 4-byte integers, 1-byte characters, and 4-byte alignment for both integers and floating-point numbers. Suppose further that we plan to use contiguous row-major layout for multidimensional arrays, that we do not wish to reorder fields of records or pack either records or arrays, and that we will assume without checking that all array subscripts are in bounds.

(a) Consider the following variable declarations.

```
A : array [1..10, 10..100] of real
i : integer
x : real
```

Show the code that our compiler should generate for the following assignment: `x := A[3,i]`. Explain how you arrived at your answer.

Answer: The address of A[3,i] is $\&A + (3 - L_1)S_1 + (i - L_2)S_2$, where $\&A$ is the address of A, $L_1 = 1$, $L_2 = 10$, $S_2 = 8$, and $S_1 = (100 - 10 + 1)S_2 = 728$. Simple algebra transforms this into $(\&A + 2S_1 - 10S_2) + (S_2 i) = (\&A + 1376) + 8i$. The first parenthesized expression is a compile-time constant. Straightforward code would look something like this:

```
r1 := i
r1 <<= 3                -- multiply by 8
r1 += *A + 1376
r2 := *r1
x := r2
```

(b) Consider the following more complex declarations.

```
r : record
      x : integer
      y : char
      A : array [1..10, 10..20] of record
            z : real
            B : array [0..71] of char
j, k : integer
```

Assume that these declarations are local to the current subroutine. Note the lower bounds on indices in A; the first element is A[1,10].

Describe how r would be laid out in memory. Then show code to load r.A[2,j].B[k] into a register. Be sure to indicate which portions of the address calculation could be performed at compile time.

Answer: Variable r would lie in the stack frame of the current subroutine. It would contain, in order, 4 bytes for x, 1 byte for y, 3 bytes of hole, and 110 contiguous elements of A. Each element of A would contain 8 bytes of z followed by 72 bytes (an even multiple of 4) of B.

To access r.A[2,j].B[k], one would perform the following calculation:

fp	+ offset of r in frame
	+ offset of A in r (i.e. 8)
	+ (2−1) × 11 × sizeof(A[x,y]) (i.e. 80)
+ j × sizeof(A[x,y])	− 10 × sizeof(A[x,y])
+ k	
= (fp + 80j + k)	+ (offset(r) + 88)

Values on the right are compile-time constants; values on the left are not known until run time. Suppose offset(r) is 12. Assembly code might look something like this:

```
r1 := j
r1 *= 80
r1 += k
r1 += 100          -- offset(r) + 88
r1 := fp[r1]       -- displacement addressing
```

8.14 Suppose A is a 10 × 10 array of (4-byte) integers, indexed from [0][0] through [9][9]. Suppose further that the address of A is currently in register r1, the value of integer i is currently in register r2, and the value of integer j is currently in register r3.

Give pseudo-assembly language for a code sequence that will load the value of A[i][j] into register r1 (a) assuming that A is implemented using (row-major) contiguous allocation; (b) assuming that A is implemented using row pointers. Each line of your pseudocode should correspond to a single instruction on a typical modern machine. You may use as many registers as you need. You need not preserve the values in r1, r2, and r3. You may assume that i and j are in bounds, and that addresses are 4 bytes long.

Which code sequence is likely to be faster? Why?

Answer:

(a)
```
r2 *:= 40
r3 <<:= 2
r1 +:= r3
r1 +:= r2
r1 := *r1
```

(b)
```
r2 <<:= 2
r1 +:= r2
r1 := *r1
r3 <<:= 2
r1 +:= r3
r1 := *r1
```

The left shifts effect multiplication by 4. The code sequence on the left is likely to be faster on a modern machine, not because it is one instruction shorter, but because it performs only one load instead of two.

8.15 Pointers and recursive type definitions complicate the algorithm for determining structural equivalence of types. Consider, for example, the following definitions:

```
type A = record
    x : pointer to B
    y : real
```

```
type B = record
    x : pointer to A
    y : real
```

The simple definition of structural equivalence given in Section 7.2.1 (expand the subparts recursively until all you have is a string of built-in types and type constructors; then compare them) does not work: we get an infinite expansion (type A = record x : pointer to record x : pointer to record x : pointer to record . . .). The obvious reinterpretation is to say two types A and B are equivalent if any sequence of field selections, array subscripts, pointer dereferences, and other operations that takes one down into the structure of A, and that ends at a built-in type, always encounters the same field names, and ends at the same built-in type when used to dive into the structure of B—and vice versa. Under this reinterpretation, A and B above have the same type. Give an algorithm based on this reinterpretation that could be used in a compiler to determine structural equivalence. (Hint: The fastest approach is due to J. Král [Krá73]. It is based on the algorithm used to find the smallest deterministic finite automaton that accepts a given regular language. This algorithm was outlined in Example 2.15; details can be found in any automata theory textbook [e.g., [HMU01]].)

Answer:

8.16 Explain the meaning of the following C declarations:

```
double *a[n];
double (*b)[n];
double (*c[n])();
double (*d())[n];
```

Answer:

```
double *a[n];       // array of n pointers to doubles
double (*b)[n];     // pointer to array of n doubles
double (*c[n])();   // array of n pointers to functions returning doubles
double (*d())[n];   // function returning pointer to array of n doubles
```

8.17 In Ada 83, pointers (`access` variables) can point only to objects in the heap. Ada 95 allows a new kind of pointer, the `access all` type, to point to other objects as well, provided that those objects have been declared to be `aliased`:

```
type int_ptr is access all Integer;
foo : aliased Integer;
ip : int_ptr;
...
ip := foo'Access;
```

The `'Access` attribute is roughly equivalent to C's "address of" (`&`) operator. How would you implement `access all` types and `aliased` objects? How would your implementation interact with automatic garbage collection (assuming it exists) for objects in the heap?

Answer: The purpose of an `access all` pointer is to refer to an object that is not dynamically allocated, and for which garbage collection is therefore not required. Dangling references could be a problem, however, if an `access all` pointer were to outlive the object to which it refers. The Ada standard avoids this problem by insisting that the scope of a referred-to object contain the scope of the pointer's type. This rule can be overridden by explicit programmer command, in which case locks and keys might prove useful: the `aliased` keyword in the variable declaration would alert the compiler to the need to provide space in the object for a lock; the `all` in a pointer declaration would indicate the need to provide space for a matching key, and to generate appropriate dynamic semantic checks.

Note that even with the scope rule in place, the `aliased` keyword alerts the compiler to the need to align the object, and to include any metadata normally present in dynamically allocated objects. It also allows the optimizer (code improver) to assume that *non*`aliased` variables are never subject to modification through pointers. The Ada 95 designers considered the possibility of allowing all pointers to refer to non-dynamically allocated objects, but opted to include the `all` modifier so that implementations could employ special optimizations for traditional heap-only pointers (e.g., representing them with a smaller number of bits when the heap is of limited size—see Section 3.7.1 of the *Ada 95 Rationale*, available on-line).

8.18 As noted in Section 8.5.2, Ada 95 forbids an `access all` pointer from referring to any object whose lifetime is briefer than that of the pointer's type. Can this rule be enforced completely at compile time? Why or why not?

Answer:

8.19 In much of the discussion of pointers in Section 8.5, we assumed implicitly that every pointer into the heap points to the *beginning* of a dynamically allocated block of storage. In some languages, including Algol 68 and C, pointers may also point to data *inside* a block in the heap. If you were trying to implement dynamic semantic checks for dangling references or, alternatively, automatic garbage collection (precise or conservative), how would your task be complicated by the existence of such "internal pointers"?

Answer:

8.20 (a) Occasionally one encounters the suggestion that a garbage-collected language should provide a `delete` operation as an optimization: by explicitly `delete`-ing objects that will never be used again, the programmer might save the garbage collector the trouble of finding and reclaiming those objects automatically, thereby improving performance. What do you think of this suggestion? Explain.

Answer: While this suggestion might indeed improve performance, it runs the risk of creating dangling pointers. It might safely be used in conjunction with locks and keys.

(b) Alternatively, one might allow the programmer to "tenure" an object, so that it will never be a candidate for reclamation. Is this a good idea?

Answer: Tenuring fits well with generational garbage collection: a tenured object can be moved from the nursery to the oldest portion of the heap, where it will be considered only if the collector runs out of memory.

8.21 In Example 8.52 we noted that functional languages can safely use reference counts since the lack of an assignment statement prevents them from introducing circularity. This isn't strictly true; constructs like the Lisp `letrec` can also be used to make cycles, so long as uses of circularly defined names are hidden inside `lambda` expressions in each definition:

```
(define foo
  (lambda ()
    (letrec ((a (lambda(f) (if f #\A b)))
             (b (lambda(f) (if f #\B c)))
             (c (lambda(f) (if f #\C a))))
      a)))
```

Each of the functions `a`, `b`, and `c` contains a reference to the next:

```
((foo) #t)                    ⟹ #\A
(((foo) #f) #t)               ⟹ #\B
((((foo) #f) #f) #t)          ⟹ #\C
(((((foo) #f) #f) #f) #t)     ⟹ #\A
```

How might you address this circularity without giving up on reference counts?

Answer: In a pure functional language data is immutable once created. If cycles can be created only with `letrec` or its equivalent, then we can tag them as cycles at creation time, and keep a single reference count for the group.

8.22 Here is a skeleton for the standard quicksort algorithm in Haskell:

```
quicksort [] = []
quicksort (a : l) = quicksort [...] ++ [a] ++ quicksort [...]
```

The `++` operator denotes list concatenation (similar to `@` in ML). The `:` operator is equivalent to ML's `::` or Lisp's `cons`. Show how to express the two elided expressions as list comprehensions.

Answer:

```
quicksort [] = []
quicksort (a : l) = quicksort [i | i <- l, i < a] ++
                    [a] ++ quicksort [i | i <- l, i >= a]
```

IN MORE DEPTH

8.23 In Example 6.70 we described a programming idiom in which an iterator takes a "loop body" function as argument, and applies it to every element of a given container or set. Show how to use this idiom in ML to apply a function to every element of the tree in Example 11.39. Write versions of your iterator for preorder, inorder, and postorder traversals.

Answer:

8.24 Show how unions can be used in C to interpret the bits of a value of one type as if they represented a value of some other type. Explain why the same technique does not work in Ada. After consulting an Ada manual, describe how an `unchecked` pragma can be used to get around the Ada rules.

Answer:

8.25 Are variant records a form of polymorphism? Why or why not?

Answer: Not really. They do let you write code that pays attention only to the "important" parts of an object, but they don't let you write code and then, later, use it for a type you didn't envision when you wrote it.

8.26 Learn the details of variant records in Pascal.

(a) You may have noticed that the language does not allow you to pass the tag field of a variant record to a subroutine by reference. Why not?

Answer: Because the subroutine could modify the tag, thereby changing the set of valid fields within the variant (and effectively rendering the new set of fields uninitialized), in a way that is very hard to catch at run time. The checking problem is compounded by the possibility of aliases: a subroutine may be able to access a variant record as a non-local variable. If the tag of that record is passed as a `var` parameter, a compiler that wants to ensure type safety must generate code that recognizes the alias and changes the accessibility of the record's fields.

(b) Explain how you might implement dynamic semantic checks to catch references to uninitialized fields of a tagged variant record. Changing the value of the tag field should cause all fields of the variant part of the record to become uninitialized. Suppose you want to avoid adding flag fields within the record itself (e.g., to avoid changing the offsets of fields in a systems program). How much harder is your task?

Answer:

(c) Explain how you might dynamic semantic checks to catch references to uninitialized fields of an *untagged* variant record. Any assignment to a field of a variant should cause all fields of other variants to become uninitialized. Any assignment that changes the record from one variant to another should also cause all other fields of the new variant to be uninitialized. Again, suppose you want to avoid adding flag fields within the untagged record itself. How much harder is your task?

Answer:

8.27 We noted in Section C-8.1.3 that Ada requires the variant portions of a record to occur at the end, to save space when a particular record is constrained to have a comparatively small variant part. Could a compiler rearrange fields to achieve

the same effect, without the restriction on the declaration order of fields? Why or why not?

Answer: Answer: no, or at least not easily. Consider nested records. If two or more fields of the outer record contain variant parts, you can't do the arrangement while still keeping fields of nested records adjacent.

8.28 In Example 8.52 we noted that reference counts can be used to reclaim tombstones, failing only when the programmer neglects to manually delete the object to which a tombstone refers. Explain how to leverage this observation to catch memory leaks at run time. Does your solution work in all cases? Explain.

Answer: If the reference count in a tombstone goes to zero but the object to which it refers has not been reclaimed, then it will never be *possible* to reclaim it, and the program has leaked storage. If there are circular references, however, and no external reference into a cycle, then all tombstones in the cycle will have nonzero reference counts, and we will fail to detect the leak.

8.29 In Section 8.5.3 we introduced the notion of *smart pointers* in C++. Learn how these are implemented, and write an explanation. Discuss the relationship to tombstones.

Answer:

8.30 Rewrite Example C-8.80 using `fgets`, `strtol`, `strtod`, etc. (read the `man` pages), so that it is guaranteed not to result in buffer overflow.

Answer:

8.31 The output routines of several languages (e.g., `println` in Swift) give special treatment to ends of lines. By contrast, C's `printf` does not; it treats newlines and carriage returns the same as any other character. What are the comparative advantages of these approaches? Which do you prefer? Why?

Answer:

9 Subroutines and Control Abstraction

9.9 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

9.1 Describe as many ways as you can in which functions in imperative programming languages differ from functions in mathematics.

Answer: A mathematical function maps values from its domain set(s) to values in its range set, deterministically. A program function can interact with the rest of the world in much more complicated ways. It may yield values that are non-deterministic or that depend on things other than its actual parameters (e.g., static or global variables, or input from the "real world"). It may also alter its environment through side effects. Side effects include not only changes to global variables and output to physical devices, but also changes to parameters that are passed by modes other than value.

In a more abstract sense, program functions are circumscribed by the notions of computability and time and space complexity. Perfectly well-defined mathematical functions can be intractable or undecidable.

9.2 Consider the following code in C++:

```
class string_map {
    string cached_key;
    string cached_val;
    const string complex_lookup(const string key);
        // body specified elsewhere
```

```
public:
    const string operator[](const string key) {
        if (key == cached_key) return cached_val;
        string rtn_val = complex_lookup(key);
        cached_key = key;
        cached_val = rtn_val;
        return rtn_val;
    }
};
```

Suppose that `string_map::operator[]` contains the only call to `complex_lookup` anywhere in the program. Explain why it would be unwise for the programmer to expand that call textually in-line and eliminate the separate function.

Answer: Because this would be likely to inhibit automatic in-line expansion of `operator[]`. Presumably `string_map` caches the previously looked-up value because repeat calls are common. If `operator[]` is expanded in-line, we will save a significant fraction of the run-time of each such repeat call.

9.3 Using your favorite language and compiler, write a program that can tell the order in which certain subroutine parameters are evaluated.

Answer: Here's an example in C:

```
#include <stdio.h>

int equal(int i, int j) {
    return i == j;
}

int inc(int* p) {
    return (*p)++;
}

int i = 1;

int main() {
    printf("%s parameter was evaluated first\n",
           equal(i, inc(&i)) ? "First" : "Second");
}
```

Note that `inc` returns the value of `*p` prior to the increment. If the first argument in `main`'s call to `equal` is evaluated first, both arguments will evaluate to 1. If the second argument is evaluated first, the first argument will evaluate to 2.

9.4 Consider the following (erroneous) program in C:

```
void foo() {
    int i;
    printf("%d ", i++);
}
```

```
int main() {
    int j;
    for (j = 1; j <= 10; j++) foo();
}
```

Local variable `i` in subroutine `foo` is never initialized. On many systems, however, the program will display repeatable behavior, printing `0 1 2 3 4 5 6 7 8 9`. Suggest an explanation. Also explain why the behavior on other systems might be different, or nondeterministic.

Answer: In most language implementations, the activation records for different instances of `foo` will occupy the same space in the stack. Local variable `i` is never initialized (that's why the program is erroneous), but if the stack space has not been used for anything else in the meantime it may "inherit" a value from the previous instance of the routine. If the stack is created from space filled by zeros by the operating system, and if the space occupied by `foo`'s activation record is not used for anything else before the first time `foo` is called, then `i` will start with the value zero in the first iteration of the loop.

9.5 The standard calling sequence for the c. 1980 Digital VAX instruction set employed not only a stack pointer (`sp`) and frame pointer (`fp`), but a separate *arguments pointer* (`ap`) as well. Under what circumstances might this separate pointer be useful? In other words, when might it be handy not to have to place arguments at statically known offsets from the `fp`?

Answer: There are several possibilities. The standard calling sequence on the VAX (supported by the `calls` and `callg` instructions) saves registers into the stack below the callee's frame pointer. The registers to be saved are specified by a bit mask in the callee's code. If a subroutine has more than one entry point (as it might, conceivably, if sometimes called through a closure), the number of saved registers—and hence the offset between the `fp` and stack-allocated arguments—may not be a compile-time constant. The VAX also word-aligns the `fp` and `sp` as a side effect of a call. This will insert 0–3 bytes of garbage into the stack, depending on previous alignment at run time.

The `calls` and `callg` instructions vary in the location of arguments: `calls` puts them in the stack; `callg` takes an extra address specifier that indicates their location, which may be anywhere in the current memory space. Non-stack arguments were never widely used on the VAX, but one could imagine them saving time for calls to nonrecursive routines with large numbers of read-only default parameters: a partially initialized argument list in static memory would avoid the overhead of pushing the default parameters on every call.

Bershad [BALL90] describes a clever use of the arguments pointer for fast remote procedure calls across address space boundaries.

9.6 Write (in the language of your choice) a procedure or function that will have four different effects, depending on whether arguments are passed by value, by reference, by value/result, or by name.

Answer: Consider the following function and call:

```
procedure f(x, y, z)
    x := x + 1
```

```
    y := z
    z := z + 1
. . .
i := 1; a[1] := 10; a[2] := 11
f(i, a[i], i)
```

Using call by value the final values of i and a are unchanged. Using call by value/result the final values are i = 2, a = [1, 11]. (Some implementations might try to flag the call as an error since if x and z in f end up with distinct values there's a problem with what value to copy back to i. As written, however, the question has a well-defined answer, since x and z don't end up with distinct values.) With call by reference i = 3, a = [2, 11]. With call by name i = 3, a = [10, 2].

9.7 Consider an expression like `a + b` that is passed to a subroutine in Fortran. Is there any semantically meaningful difference between passing this expression as a reference to an unnamed temporary (as Fortran does) or passing it by value (as one might, for example, in Pascal)? That is, can the programmer tell the difference between a parameter that is a value and a parameter that is a reference to a temporary?

Answer: The programmer may think of these two options differently, but the behavior will be the same.

9.8 Consider the following subroutine in Fortran 77:

```
subroutine shift(a, b, c)
integer a, b, c
a = b
b = c
end
```

Suppose we want to call `shift(x, y, 0)` but we don't want to change the value of `y`. Knowing that built-up expressions are passed as temporaries, we decide to call `shift(x, y+0, 0)`. Our code works fine at first, but then (with some compilers) fails when we enable optimization. What is going on? What might we do instead?

Answer: Our optimizer recognizes that `y+0 .eq. y`, skips the addition, and passes plain `y` instead. A safer strategy is to create a local variable `z`, assign it the value of `y`, and pass it instead.

9.9 In some implementations of Fortran IV, the following code would print a 3. Can you suggest an explanation? How do you suppose more recent Fortran implementations get around the problem?

```
c   main program
    call foo(2)
    print*, 2
    stop
    end
    subroutine foo(x)
        x = x + 1
        return
    end
```

Answer: On a machine without an immediate addressing mode, constants must be kept in memory. For the program shown, the compiler would reserve a location in global memory to hold the value 2. When passing that value to a subroutine, since Fortran uses call-by-reference exclusively, some compilers would simply use the address of the constant. Unfortunately, if the subroutine modified the parameter, any code later in the program that tried to use the same constant would see the modified value. The solution (adopted by all modern Fortran compilers) is straightforward: create a temporary copy and pass its address instead.

9.10 Suppose you are writing a program in which all parameters must be passed by name. Can you write a subroutine that will swap the values of its actual parameters? Explain. (Hint: Consider mutually dependent parameters like `i` and `A[i]`.)

Answer: No, it does not appear to be possible to write a fully general `swap` routine with call-by-name parameters. For a discussion of the problem, see the article by A. C. Fleck [Fle76].

9.11 Can you write a `swap` routine in Java, or in any other language with only call-by-sharing parameters? What exactly should `swap` *do* in such a language? (Hint: Think about the distinction between the object to which a variable refers and the value [contents] of that object.)

Answer: The natural code

```
public static void swap(Object a, Object b) {
    Object t = b;
    b = a;
    a = t;
}
```

does not work. It makes formal parameters `a` and `b` point at each other's objects, but this does not change the actual parameters. There is no way with call by sharing to swap the objects referred to by the actual parameters. One *can* write class-specific routines that swap the *values* of the objects to which `a` and `b` refer:

```
class Foo {
    int v;
    public int getVal() { return v; }
    public void setVal(int n) { v = n; }
    public Foo(int n) { v = n; }
}
...
public static void swap(Foo a, Foo b) {
    int t = b.getVal();
    b.setVal(a.getVal());
    a.setVal(t);
}
```

This doesn't work in general because Java doesn't provide a way to change the values of arbitrary objects.

9.12 As noted in Section 9.3.1, `out` parameters in Ada 83 can be written by the callee but not read. In Ada 95 they can be both read and written, but they begin their life uninitialized. Why do you think the designers of Ada 95 made this change? Does it have any drawbacks?

Answer: `Out` parameters in Ada 83 suffer from a problem similar to that described for function returns in Section 9.3.4: because the parameter is write-only, the programmer must employ a temporary local variable in incremental computations. Ada 95 eliminates this restriction. The only real cost is a possible perceived loss in the symmetry of parameter mode semantics.

9.13 Taking a cue from Ada, Swift provides an `inout` parameter mode. The language manual does not specify whether `inout` parameters are to be passed by reference or value-result. Write a program that determines the implementation used by your local Swift compiler.

Answer:

```
var x : Int = 1

func peek_at(inout y : Int) {
    println("  b: \(x)")
    y++
    println("  c: \(x)")
    y++
    println("  d: \(x)")
}

println("a: \(x)")
peek_at(&x)
println("e: \(x)")
```

As of June 2015, with `swiftc` version 1.1, this code prints

```
a: 1
  b: 1
  c: 2
  d: 3
e: 3
```

This would seem to confirm that the compiler is passing by reference.

9.14 Fields of *packed* records (Example 8.8) cannot be passed by reference in Pascal. Likewise, when passing a subrange variable by reference, Pascal requires that all possible values of the corresponding formal parameter be valid for the subrange:

```
type small = 1..100;
     R = record x, y : small; end;
     S = packed record x, y : small; end;
var  a : 1..10;
     b : 1..1000;
```

```
     c : R;
     d : S;
procedure foo(var n : small);
begin
    n := 100;
    writeln(a);
end;
...
a := 2;
foo(b);     (* ok *)
foo(a);     (* static semantic error *)
foo(c.x);   (* ok *)
foo(d.x);   (* static semantic error *)
```

Using what you have learned about parameter-passing modes, explain these language restrictions.

Answer: The first restriction avoids alignment problems. If fields of packed arrays could be passed by reference, `foo` would have to use run-time checks to determine whether to access `n` with a single (aligned) load or store, or whether to use a slower multi-instruction sequence.

The second restriction prevents `foo` from assigning an out-of-range value into `n`, and then reading it out of `a`. Ada has no such restriction: scalars are passed by copying, and range constraints on numeric parameters are checked when the subroutine returns. An Ada program syntactically similar to our Pascal example (with `n` passed with mode `in out`) would pass successfully through the compiler, but the second return from `foo` would result in a dynamic semantic error.

9.15 Consider the following declaration in C:

```
double(*foo(double (*)(double, double[]), double)) (double, ...);
```

Describe in English the type of `foo`.

Answer: Recall the rule for C: start at the identifier; work as far right as possible, without escaping parentheses, then work as far left as possible without escaping parentheses, then move out one level and repeat.

Moving right from `foo` we find an argument list, so `foo` is a function, specifically a function of two parameters, neither of which is named. The first parameter to `foo` is a pointer to a function. For the sake of clarity, call that function *F*; if it had had a name the name would have appeared between the `*` and the following right parenthesis. *F*'s first argument is a double; its second argument is an array of doubles, of unspecified length. The second parameter to `foo` is also a double.

Moving left from `foo` we find another `*`, so `foo` returns a pointer. To figure out the type of the thing to which it points, we move out a level of parentheses. Moving right from there we find another argument list, so `foo` returns a pointer to a function. That function, call it *G*, takes variable-length argument lists: the first parameter is a double; the others (zero or more of them) can be of any type. Finally, moving left, *G* returns a double.

Because no function body is provided, this is a declaration of `foo`, but not a definition. The body must be provided elsewhere.

9.16 Does a program run faster when the programmer leaves optional parameters out of a subroutine call? Why or why not?

Answer: Not with most machines and compilers. The code for a call with optional parameters is exactly the same as the code for a call in which all parameters are specified explicitly, with their default values. If one used a separate arguments pointer (as supported in hardware on the VAX [see Exercise 9.5]) it might be possible, for nonrecursive routines, to use a static build area, prepared at compile time, that contains precomputed default values for optional parameters.

9.17 Why do you suppose that variable-length argument lists are so seldom supported by high-level programming languages?

Answer: Because they aren't type safe in languages with static typing, at least in the general form. Several languages allow lists in which all trailing arguments have the same (declared) type.

9.18 Building on Exercise 6.35, show how to implement exceptions using `call-with-current-continuation` in Scheme. Model your syntax after the `handler-case` of Common Lisp. As in Exercise 6.35, you will probably need `define-syntax` and `dynamic-wind`.

Answer:

```
(define handlers '())    ; stack initially empty

(define-syntax catch-list
  (syntax-rules ()
    ((catch-list) '())
    ((catch-list (tag () item)) (list (cons 'tag (lambda (_) item))))
    ((catch-list (tag (arg) item)) (list (cons 'tag (lambda (arg) item))))
    ((catch-list (tag () item) more ...)
     (cons (cons 'tag (lambda (_) item)) (catch-list more ...)))
    ((catch-list (tag (arg) item) more ...)
     (cons (cons 'tag (lambda (arg) item)) (catch-list more ...)))))

(define-syntax handler-case
  (syntax-rules ()
    ((run body handler ...)
     (let* ((helper (lambda (c)
                      (c (dynamic-wind
                           (lambda ()
                             (set! handlers (cons c handlers)))
                           (lambda () (list #t body))
                           (lambda ()
                             (set! handlers (cdr handlers)))))))
            (result (call-with-current-continuation helper))
            (val (cadr result)))
```

```
      (if (car result)
          val
          (let* ((catches (catch-list handler ...))
                 (match (assv val catches))
                 (default (assv 'else catches)))
            (if match
                ((cdr match) (caddr result))
                (if default
                    ((cdr default) (caddr result))
                    (error val (caddr result))))))))))

(define error
  (lambda (tag val)
    (if (null? handlers)
        (begin (display "uncaught exception ") (display tag) (newline))
        (let ((c (car handlers)))
          (c (list #f tag val))))))

(define test
  (lambda (n)
    (handler-case
       (cond
         ((= n 1) (error 'foo "ouch"))
         ((= n 2) (error 'bar 5))
         ((= n 3) (error 'glarch '()))
         (else (+ n 1)))
     (bar (i) (+ i 1))
     (glarch () (begin (display "glarch") (newline) 0))
     (else (s) (begin (display s) (newline) 0)))))

(test 0) ⟹ 1
(test 1) ⟹ 0  (and prints "ouch")
(test 2) ⟹ 6
(test 3) ⟹ 0  (and prints "glarch")
(test 4) ⟹ 5
```

9.19 Given what you have learned about the implementation of structured exceptions, describe how you might implement the nonlocal `goto`s of Pascal or the label parameters of Algol 60 (Section 6.2). Do you need to place any restrictions on how these features can be used?

Answer:

9.20 Describe a plausible implementation of C++ destructors or Java `try...finally` blocks. What code must the compiler generate, at what points in the program, to ensure that cleanup always occurs when leaving a scope?

Answer: For dynamically allocated objects in C++, destruction occurs when the object is `deleted`. For derived classes, code for destructor begins with a call to the destructors of parent classes. Destructors for global and static objects are called at the end of execution. Typically, every separately compiled module contains a block of code to call the destructors of all globals and statics in that module. The linker arranges for the main program to call this code just prior to program termination.

Destructors for objects declared in a local scope are a bit more complicated. Though C++ lacks a `finally` construct, the compiler generates its equivalent for any scope containing objects with destructors.

Though syntactically similar to a `catch` block in languages like Java, a `finally` block is implemented somewhat differently. In the wake of an exception, control continues immediately after the matching `catch` block. By contrast, a `finally` block is executed *on the way* to some other destination. Typically, it is implemented as a hidden subroutine, called immediately before any control transfer that would leave the corresponding `try` block. Detailed information on the implementation of Java `finally` blocks can be found in documentation for the Java Virtual Machine [LY99, Secs. 4.9.6 and 7.13].

Code can leave a `try` block in Java in several ways: it can fall off the bottom of the code; it can execute a `return`, `break`, `continue`, or `throw` statement; or it can suffer an exception thrown in some called method. These are all the possibilities (the list may be different in other languages). In every case, the compiler knows that control is about to leave the `try` block. It simply generates a call to the subroutine representing the `finally` block immediately before the normal transfer. The only tricky case occurs in exception propagation. As we saw in Section 9.4.3, however, the typical implementation of a `try` block has a hidden "catch-all" handler that reraises any exception not handled locally. Calls to `finally` blocks are naturally inserted immediately prior to the reraise.

9.21 Use threads to build support for true iterators in Java. Try to hide as much of the implementation as possible behind a reasonable interface. In particular, hide any uses of `new thread`, `thread.start`, `thread.join`, `wait`, and `notify` inside implementations of routines named `yield` (to be called by an iterator) and in the standard Java `Iterator` interface routines (to be called in the body of a loop). Compare the performance of your iterators to that of the built-in iterator objects (it probably won't be good). Discuss any weaknesses you encounter in the abstraction facilities of the language.

Answer:

9.22 In Common Lisp, multilevel returns use `catch` and `throw`; exception handling in the style of most other modern languages uses `handler-case` and `error`. Show that the distinction between these is mainly a matter of style, rather than expressive power. In other words, show that each facility can be used to emulate the other.

Answer:

9.23 Compile and run the program in Figure 9.6. Explain its behavior. Create a new version that behaves more predictably.

```
#include <signal.h>
#include <stdio.h>
#include <string.h>

char* days[7] = {"Sunday", "Monday", "Tuesday",
                 "Wednesday", "Thursday", "Friday", "Saturday"};
char today[10];

void handler(int n) {
    printf(" %s\n", today);
}

int main() {
    signal(SIGTSTP, handler);       // ^Z at keyboard
    for(int n = 0; ; n++) {
        strcpy(today, days[n%7]);
    }
}
```

Figure 9.6 A problematic program in C to illustrate the use of signals. In most Unix systems, the `SIGTSTP` signal is generated by typing control-Z at the keyboard.

Answer: In the main loop of the program, we need to disable and reenable signals around the string copy operation:

```
for(int n = 0; ; n++) {
    sigprocmask(SIG_BLOCK, &tstp, 0);
    strcpy(today, days[n%7]);
    sigprocmask(SIG_UNBLOCK, &tstp, 0);
}
```

9.24 In C#, Java, or some other language with thread-based event handling, build a simple program around the "pause button" of Examples 9.51–9.54. Your program should open a small window containing a text field and two buttons, one labeled "pause", the other labeled "resume". It should then display an integer in the text field, starting with zero and counting up once per second. If the pause button is pressed, the count should suspend; if the resume button is pressed, it should continue.

Note that your program will need at least two threads—one to do the counting, one to handle events. In Java, the JavaFX package will create the handler thread automatically, and your main program can do the counting. In C#, some existing thread will need to call `Application.Run` in order to become a handler thread. In this case you'll need a second thread to do the counting.

Answer: Here's a solution in C#.

```
using System;
using System.Threading;
using Gtk;
```

```
class Counter {
    volatile bool running = true;
    Label myLabel;
    Window myWin;
    public Counter(Window w, Label l) {
        myWin = w;
        myLabel = l;
    }
    public void Run() {
        int n = 0;
        while(true) {
            myLabel.Text = n.ToString();
            Thread.Sleep(1000);             // milliseconds
            myWin.ShowAll();
            if (running) n++;
        }
    }
    public void Pause() {
        running = false;
    }
    public void Resume() {
        running = true;
    }
}

class Demo : Window {
    Thread myThread;
    Counter myCount;

    public Demo(string title) : base(title) {
        Resize(240,120);
        DeleteEvent += delegate(object sender, DeleteEventArgs a) {
            myThread.Abort();
            myThread.Join();
            Application.Quit();
            a.RetVal = true;        // not strictly required
        };

        Label myLabel = new Label();
        Button pauseButton = new Button("pause");
        pauseButton.Clicked += delegate(object sender, EventArgs a) {
            myCount.Pause();
        };
```

```
        Button resumeButton = new Button("resume");
        resumeButton.Clicked += delegate(object sender, EventArgs a) {
            myCount.Resume();
        };

        VBox myBox = new VBox();
        myBox.PackStart(myLabel, false, false, 10);
        myBox.PackStart(pauseButton, false, false, 10);
        myBox.PackStart(resumeButton, false, false, 10);

        Add(myBox);

        myCount = new Counter(this, myLabel);

        // create worker thread to do the counting:
        myThread = new Thread(new ThreadStart(myCount.Run));
        myThread.Start();
    }
}

public class EventDemo {
    public static void Main() {
        Application.Init();

        Demo myWin = new Demo("Event demo");
        myWin.ShowAll();

        Application.Run();      // become an event handler thread
    }
}
```

9.25 Extend your answer to the previous problem by adding a "clone" button. Pushing this button should create an additional window containing another counter. This will, of course, require additional threads.

Answer:

IN MORE DEPTH

9.26 Suppose you wish to minimize the size of closures in a language implementation that uses a display to access nonlocal objects. Assuming a language like Pascal or Ada, in which subroutines have limited extent, explain how an appropriate display for a formal subroutine can be calculated when that routine is finally called, starting with only (1) the value of the frame pointer, saved in the closure at the time that the closure was created, (2) the subroutine return addresses found in

the stack at the time the formal subroutine is finally called, and (3) static tables created by the compiler. How costly is your scheme?

Answer:

9.27 Elaborate on the reasons why even parameters passed in registers may sometimes need to have locations in the stack. Consider all the cases in which it may not suffice to keep a parameter in a register throughout its lifetime.

Answer: (1) If we run out of registers we may need to spill some values. (2) If there are subroutines nested inside us, they may access our parameters directly. If we call such a routine we need to make sure that an up-to-date value can be found in memory. (3) If we ever create a pointer to one of our parameters (as we can, for example, in C), then we need to be sure that an up-to-date value can be found in memory any time the pointer may be used.

9.28 Most versions of the C library include a function, `alloca`, that dynamically allocates space within the current stack frame.[2] It has two advantages over the usual `malloc`, which allocates space in the heap: it's usually very fast, and the space it allocates is reclaimed automatically when the current subroutine returns. Assuming the programmer *wants* deallocation to happen then, it's convenient to be able to skip the explicit `free` operations. How might you implement `alloca` in conjunction with the calling conventions of our various case studies?

Answer: The basic idea is for `alloca` to update the stack pointer to push the newly allocated space. This doesn't work, of course, if the stack pointer is being used as the base address for the stack frame, or if the call occurs in the middle of building an argument list, for which the stack pointer must serve as the base. LLVM addresses the first problem by never eliding the frame pointer in any subroutine that includes a call to `alloca`. This implies that knowledge of `alloca` must be built into the compiler itself, but similar special-casing occurs for several other routines. Given the need to track calls to `alloca`, LLVM actually includes the entire implementation in the compiler; is uses no actual library code. The second problem (calls within argument lists) is handled by performing the allocation early, before assembly of the argument list begins.

9.29 Explain how to extend the conventions of Figure C-9.9 and Section C-9.2.2 to accommodate arrays whose bounds are not known until elaboration time (as discussed in Section 8.2.2). What ramifications does this have for the use of separate stack and frame pointers?

9.30 In all three of our case studies, stack-based arguments were placed into the argument build area in "reverse" order, with the lowest-numbered argument at the top. Explain why this is important. (Hint: Consider subroutines with variable numbers of arguments, as discussed in Section 9.3.3.)

Answer: Because C supports subroutines with a variable number of arguments, a subroutine cannot in general know how many arguments there will be in any given call. Putting the first argument at a known offset from the `fp` ensures that it, at least, can be found. Programming

---

2 Unfortunately, `alloca` is not POSIX compliant, and implementations vary greatly in their semantics and even in details of the interface. Portable programs are wise to avoid this routine.

conventions typically require that it be possible to deduce the number and types of trailing arguments starting with the first few.

9.31 How would you implement nested subroutines as parameters on a machine that doesn't let you execute code in the stack? Can you pass a simple code address, or do you require that closures be interpreted at run time?

Answer: If we wish to represent a closure with a simple code address, then the code at that address must be able, at run time, to achieve the effect of calling the right subroutine with the right (dynamically created) static link. One could imagine creating, at compile time, a fixed number of "trampoline" routines, each of which interprets a closure with which it is statically associated, but this fails if the program needs an unbounded number of closures. In general, representing closures with a code pointer requires the ability to generate code at run time. If the stack is nonexecutable, this code might be allocated in a special, executable heap.

9.32 If you have read the rest of Chapter 9, you may have noticed that the term "trampoline" is also used in conjunction with the implementation of signal handlers (Section 9.6.1). What is the connection (if any) between these uses of the term?

Answer: Both are used to describe code that somehow "patches things up" in order to call a subroutine from a context in which the usual calling sequence would not suffice.

9.33 Explain how you might implement `setjmp` and `longjmp` on a SPARC.

Answer:

9.34 Following the code in Figure 6.5, and assuming a single-stack implementation of iterators, trace the contents of the stack during the execution of a `for` loop that iterates over all nodes of a complete, 3-level (6-node) binary tree.

Answer:

9.35 Build a preorder iterator for binary trees in Java, C#, or Python. Do not use a true iterator or an explicit stack of tree nodes. Rather, create nested iterator objects on demand, linking them together as a C# compiler might if it were building the iterator object equivalent of a true preorder iterator.

Answer:

9.36 One source of inaccuracy in the traffic simulation of Section C-9.5.4 has to do with the timing at traffic signals. If a signal is currently green in the `EW` direction, but the queue of waiting cars is empty, the `signal` coroutine will go to sleep until `current_time + EW_duration`. If a car arrives before the coroutine wakes up again, it will needlessly wait. Discuss how you might remedy this problem.

Answer:

10 Data Abstraction and Object Orientation

10.10 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

10.1 Some language designers argue that object orientation eliminates the need for nested subroutines. Do you agree? Why or why not?

Answer: This is open to debate. Here is an argument for "yes": There are two main reasons for nested subroutines: information hiding and state sharing. Objects do both. Objects are actually better at information hiding, since they allow a subroutine to be shared by several other subroutines (rather than just the one inside of which it is nested), without being visible to the rest of the program. Similarly, the data members of an object are available to all the members, regardless of nesting. On the other hand, there may be cases in which it is useful to nest subroutines (and share data among them) at more than one level, or to define an auxiliary subroutine inside another subroutine, without having to create an object. There is also strong anecdotal evidence to suggest that nesting enhances the expressive power of first-class functions in functional programming languages. Programs written in CLOS certainly make extensive use of nested functions.

10.2 Design a class hierarchy to represent syntax trees for the CFG of Figure 4.5. Provide a method in each class to return the value of a node. Provide constructors that play the role of the `make_leaf`, `make_un_op`, and `make_bin_op` subroutines.

Answer: Here is one possible solution in C++:

```
class node {
public:
    virtual int value() = 0;     // virtual class
};
```

1

```
class bin_op : public node {
    // no value method; still a virtual class
protected:
    node *left_operand;
    node *right_operand;
public:
    bin_op(node *l, node *r) : left_operand(l), right_operand(r) {}
};

class plus : public bin_op {
public:
    plus(node *l, node *r) : bin_op(l, r) {}
    virtual int value()
        {return left_operand->value() + right_operand->value();}
};

class minus : public bin_op {
public:
    minus(node *l, node *r) : bin_op(l, r) {}
    virtual int value()
        {return left_operand->value() - right_operand->value();}
};

class times : public bin_op {
public:
    times(node *l, node *r) : bin_op(l, r) {}
    virtual int value()
        {return left_operand->value() * right_operand->value();}
};

class div : public bin_op {
public:
    div(node *l, node *r) : bin_op(l, r) {}
    virtual int value()
        {return left_operand->value() / right_operand->value();}
};

class un_op : public node {
    // no value method; still a virtual class
protected:
    node *operand;
public:
    un_op(node *o) : operand(o) {}
};
```

```
class negate : public un_op {
public:
    negate(node *n) : un_op(n) {};
    virtual int value() {return -operand->value();}
};

class number : public node {
    int val;
public:
    number(int v) : val(v) {};
    virtual int value() {return val;}
};

/*************** TESTING *****************/

#include <iostream>
using std::cout;

int main() {
    node *a = new number(5);
    node *b = new number(4);
    node *c = new number(3);
    node *d = new number(2);
    node *e = new minus(a, b);
    node *f = new times(c, d);
    node *g = new times(c, d);
    node *r = new plus(f, g);
    cout << r->value() << "\n";     // (- (5 - 4)) + (3 * 2) = 5
}
```

Alternative formulations are possible. We could, for example, flatten the hierarchy and add an extra `operation` field (a function pointer) to classes `bin_op` and `un_op`, rather than deriving the `plus`, `minus`, `times`, `div`, and `negate` subclasses. The version shown is arguably more object-oriented.

10.3 Repeat the previous exercise, but using a variant record (union) type to represent syntax tree nodes. Repeat again using type extensions. Compare the three solutions in terms of clarity, abstraction, type safety, and extensibility.

Answer: Here is a solution in Pascal, using unions (variant records):

```
program ast(output);

type
    node_type = (plus, minus, times, divide, negate, number);
    node_ptr = ^node;
```

```
    node = record
        case tp : node_type of
            plus, minus, times, divide : (
                left_operand, right_operand : node_ptr;
            );
            negate : (
                operand : node_ptr;
            );
            number : (
                val : integer;
            );
    end;

function make_bin_op(o : node_type; l, r : node_ptr) : node_ptr;
var p : node_ptr;
begin
    new(p);
    p^.tp := o;
    p^.left_operand := l;
    p^.right_operand := r;
    make_bin_op := p;
end;

function make_negate(n : node_ptr) : node_ptr;
var p : node_ptr;
begin
    new(p);
    p^.tp := negate;
    p^.operand := n;
    make_negate := p;
end;

function make_leaf(v : integer) : node_ptr;
var p : node_ptr;
begin
    new(p);
    p^.tp := number;
    p^.val := v;
    make_leaf := p;
end;

function value(n : node_ptr) : integer;
begin
    case n^.tp of
        plus : value := value(n^.left_operand) + value(n^.right_operand);
        minus : value := value(n^.left_operand) - value(n^.right_operand);
        times : value := value(n^.left_operand) * value(n^.right_operand);
        divide : value := value(n^.left_operand) div value(n^.right_operand);
```

```
        negate : value := - value(n^.operand);
        number : value := n^.val;
    end;
end;

(*************** TESTING *****************)

var a, b, c, d, e, f, g, r : node_ptr;

begin
    a := make_leaf(5);
    b := make_leaf(4);
    c := make_leaf(3);
    d := make_leaf(2);
    e := make_bin_op(minus, a, b);
    f := make_negate(e);
    g := make_bin_op(times, c, d);
    r := make_bin_op(plus, f, g);
    writeln(value(r));                  (* (- (5 - 4)) + (3 * 2) = 5 *)
end.
```

Here is a solution in Ada 95 using type extensions instead of classes, and using generics to capture the mathematical functions performed by binary and unary operators (function pointers could also be used).

```
package node is
    type node_t is abstract tagged null record;
    type node_ptr is access all node_t'Class;
    function value(self : access node_t) return integer;
end node;

package body node is
    function value(self : access node_t) return integer is
    begin
        return 0;
    end value;
end node;

generic
    with function op(l, r : integer) return integer;
package node.bin_op is
    type bin_op_t is new node_t with private;
    type bin_op_ptr is access all bin_op_t;
    function value(self : access bin_op_t) return integer;
    procedure initialize(self : access bin_op_t; l, r : node_ptr);
```

```
private
    type bin_op_t is new node_t with record
        left_operand : node_ptr;
        right_operand : node_ptr;
    end record;
end node.bin_op;

package body node.bin_op is
    function value(self : access bin_op_t) return integer is
    begin
        return op(value(self.left_operand), value(self.right_operand));
    end value;
    procedure initialize(self : access bin_op_t; l, r : node_ptr) is
    begin
        self.left_operand := l;
        self.right_operand := r;
    end initialize;
end node.bin_op;

package node.plus is new node.bin_op("+");
package node.minus is new node.bin_op("-");
package node.times is new node.bin_op("*");
package node.divide is new node.bin_op("/");

generic
    with function op(n : integer) return integer;
package node.un_op is
    type un_op_t is new node_t with private;
    type un_op_ptr is access all un_op_t;
    function value(self : access un_op_t) return integer;
    procedure initialize(self : access un_op_t; n : node_ptr);
private
    type un_op_t is new node_t with record
        operand : node_ptr;
    end record;
end node.un_op;

package body node.un_op is
    function value(self : access un_op_t) return integer is
    begin
        return op(value(self.operand));
    end value;
    procedure initialize(self : access un_op_t; n : node_ptr) is
    begin
        self.operand := n;
    end initialize;
end node.un_op;
```

```
package node.negate is new node.un_op("-");

package node.number is
    type number_t is new node_t with private;
    type number_ptr is access number_t'Class;
    function value(self : access number_t) return integer;
    procedure initialize(self : access number_t; v : integer);
private
    type number_t is new node_t with record
        val : integer;
    end record;
end node.number;

package body node.number is
    function value(self : access number_t) return integer is
    begin
        return self.val;
    end value;
    procedure initialize(self : access number_t; v : integer) is
    begin
        self.val := v;
    end initialize;
end node.number;

with text_io; use text_io;

---------------- TESTING ----------------

procedure node_test is
package int_io is new integer_io(integer); use int_io;

a, b, c, d : number_ptr;
e : minus.bin_op_ptr;
f : negate.un_op_ptr;
g : times.bin_op_ptr;
r : plus.bin_op_ptr;

begin
    a := new number_t;
    number.initialize(a, 5);
    b := new number_t;
    number.initialize(b, 4);
    c := new number_t;
    number.initialize(c, 3);
    d := new number_t;
    number.initialize(d, 2);
```

```
    e := new minus.bin_op_t;
    minus.initialize(e, node_ptr(a), node_ptr(b));
    f := new negate.un_op_t;
    negate.initialize(f, node_ptr(e));
    g := new times.bin_op_t;
    times.initialize(g, node_ptr(c), node_ptr(d));
    r := new plus.bin_op_t;
    plus.initialize(r, node_ptr(f), node_ptr(g));
    int_io.put(value(r));              -- (- (5 - 4)) + (3 * 2) = 5 *)
end node_test;
```

The C++ and Ada programs are type-safe; the Pascal version is not. Likewise, the C++ and Ada versions do a better job of capturing abstraction. We could extend either of them to new classes of syntax tree nodes without modifying existing code. Clearly the Ada version is the most verbose of the three, partly because of our choice to use generics, partly because Ada syntax is simply "wordier," but largely because the type extension mechanism requires more syntactic noise for explicit "this" parameters, class-wide types, and "down-conversions" to parent type.

10.4 Using the C# *indexer* mechanism, create a hash table class that can be indexed like an array. (In effect, create a simple version of the `System.Collections.Hashtable` container class.) Alternatively, use an overloaded version of `operator[]` to build a similar class in C++.

Answer:

10.5 In the spirit of Example 10.8, write a *double-ended queue* (deque) abstraction (pronounced "deck"), derived from a doubly-linked list base class.

Answer:

10.6 Use templates (generics) to abstract your solutions to the previous two questions over the type of data in the container.

Answer:

10.7 Repeat Exercise 10.5 in Python or Ruby. Write a simple program to demonstrate that generics are not needed to abstract over types. What happens if you mix objects of different types in the same deque?

Answer:

10.8 When using the `list` class of Example 10.17, the typical C++ programmer will use a pointer type for generic parameter `V`, so that `list_nodes` point to the elements of the list. An alternative implementation would include `next` and `prev` pointers for the list within the elements themselves—typically by arranging for the element type to inherit from something like the `gp_list_node` class of Example 10.14. The result is sometimes called an *intrusive* list.

(a) Explain how you might build intrusive lists in C++ without requiring users to pepper their code with explicit type casts. Hint: given multiple inheritance, you will probably need to determine, for each concrete element

type, the offset within the representation of the type at which the `next` and `prev` pointers appear. For further ideas, search for information on the `boost::intrusive::list` class of the popular Boost library.

(b) Discuss the relative advantages and disadvantages of intrusive and non-intrusive lists.

Answer:

10.9 Can you emulate the inner class of Example 10.22 in C# or C++? (Hint: You'll need an explicit version of Java's hidden reference to the surrounding class.)

Answer: Here is a solution in C#:

```
class Outer {
    public int n;
    class Inner {
        Outer parent;               // explicit reference
        public Inner(Outer o) {     // passed to constructor
            parent = o;             // and saved for future use
        }
        public void bar() {
             parent.n = 1;          // like here
        }
    }
    Inner i;
    public Outer() {
        i = new Inner(this);        // note argument!
    }
    public void foo() {
        n = 0;
        Console.WriteLine(n);       // prints 0
        i.bar();
        Console.WriteLine(n);       // prints 1
    }
}
```

10.10 Write a package body for the list abstraction of Figure 10.2.

Answer:

10.11 Rewrite the list and queue abstractions in Eiffel, Java, and/or C#.

Answer:

10.12 Using C++, Java, or C#, implement a `Complex` class in the spirit of Example 10.25. Discuss the time and space tradeoffs between maintaining all four values (x, y, ρ, and θ) in the state of the object, or keeping only two and computing the others on demand.

Answer: Here's a C# solution that maintains (only) x and y explicitly, but allows any of the four to be accessed or updated:

```
class complex {
    double x;                   // x and y (lower case) are private
    double y;
    public double X {
        get { return x; }       // presence of get accessor and optional
        set { x = value; }      // set accessor means that X is a property
    }
    public double Y {           // analogous to X
        get { return y; }
        set { y = value; }
    }
    public double Rho {
        get { return Math.Sqrt(x*x + y*y); }
        set {
            double theta = Theta;
            x = value * Math.Cos(theta);
            y = value * Math.Sin(theta);
        }
    }
    public double Theta {
        get { return Math.Atan2(y, x); }
        set {
            double rho = Rho;
            x = rho * Math.Cos(value);
            y = rho * Math.Sin(value);
        }
    }
    public complex() { x = y = 0; }
}
```

A version that maintained just ρ and θ would be symmetric; a version that maintained all four would be straightforward. Obviously the four-field version takes more space. The time tradeoff depends on the access pattern. If all four properties are frequently read but rarely written, then the four-field version will perform less computation. If writes are more common than reads, one of the two-field versions will be faster. If one of the sets of coordinates is accessed much more often than the other, it probably makes sense to represent only properties with fields.

10.13 Repeat the previous two exercises for Python and/or Ruby.

Answer:

10.14 Compare Java `final` methods with C++ nonvirtual methods. How are they the same? How are they different?

Answer: Java `final` methods, like C++ nonvirtual methods, require no indirection, and can safely be inlined. Unlike C++ nonvirtual methods, Java `final` methods cannot be overridden in a derived class. This rule allows Java to retain the guarantee about always using the appropriate method implementation when calling through a reference.

10.15 In several object-oriented languages, including C++ and Eiffel, a derived class can hide members of the base class. In C++, for example, we can declare a base class to be `public`, `protected`, or `private`:

```
class B : public A { ...
    // public members of A are public members of B
    // protected members of A are protected members of B
...
class C : protected A { ...
    // public and protected members of A are protected members of C
...
class D : private A { ...
    // public and protected members of A are private members of D
```

In all cases, `private` members of `A` are inaccessible to methods of `B`, `C`, or `D`.

Consider the impact of `protected` and `private` base classes on dynamic method binding. Under what circumstances can a reference to an object of class `B`, `C`, or `D` be assigned into a variable of type `A*`?

Answer:

```
class B : public A { ...
    // anybody can convert (assign) a B* into an A*
...
class C : protected A { ...
    // only members and friends of C or its derived classes
    // can convert (assign) a C* into an A*
...
class D : private A { ...
    // only members and friends of D itself
    // can convert (assign) a D* into an A*
```

10.16 What happens to the implementation of a class if we redefine a data member? For example, suppose we have:

```
class foo {
public:
    int a;
    char *b;
};
...
class bar : public foo {
public:
    float c;
    int b;
};
```

Does the representation of a `bar` object contain one `b` field or two? If two, are both accessible, or only one? Under what circumstances?

Answer:

10.17 Discuss the relative merits of classes and type extensions. Which do you prefer? Why?

Answer:

10.18 Building on the outline of Example 10.28, write a program that illustrates the difference between copy constructors and `operator=` in C++. Your code should include examples of each situation in which one of these may be called (don't forget parameter passing and function returns). Instrument the copy constructors and assignment operators in each of your classes so that they will print their names when called. Run your program to verify that its behavior matches your expectations.

Answer:

10.19 What do you think of the decision, in C++, C#, and Ada 95, to use static method binding, rather than dynamic, by default? Is the gain in implementation speed worth the loss in abstraction and reusability? Assuming that we sometimes want static binding, do you prefer the method-by-method approach of C++ and C#, or the variable-by-variable approach of Ada 95? Why?

Answer:

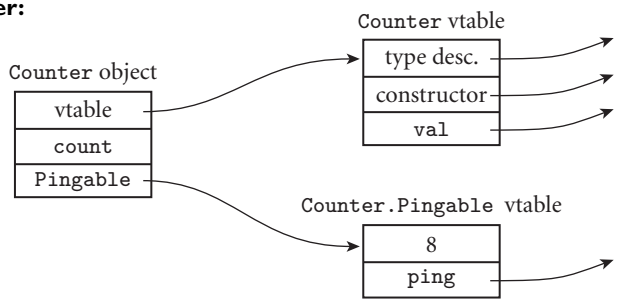
10.20 If `foo` is an abstract class in a C++ program, why is it acceptable to declare variables of type `foo*`, but not of type `foo`?

Answer:

10.21 Consider the Java program shown in Figure 10.8. Assume that this is to be compiled to native code on a machine with 4-byte addresses.

(a) Draw a picture of the layout in memory of the object created at line 15. Show all virtual function tables.

Answer:



(b) Give assembly-level pseudocode for the call to `c.val` at line 19. You may assume that the address of `c` is in register `r1` immediately before the call, and that this same register should be used to pass the hidden `this` parameter. You may ignore the need to save and restore registers, and don't worry about where to put the return value.

```
1.  interface Pingable {
2.      public void ping();
3.  }

4.  class Counter implements Pingable {
5.      int count = 0;
6.      public void ping() {
7.          ++count;
8.      }
9.      public int val() {
10.         return count;
11.     }
12. }

13. public class Ping {
14.     public static void main(String args[]) {
15.         Counter c = new Counter();
16.         c.ping();
17.         c.ping();
18.         int v = c.val();
19.         System.out.println(v);
20.     }
21. }
```

Figure 10.8 A simple program in Java.

Answer:

```
-- &c in r1
r2 = *r1          -- Counter vtable
call *(r2+8)      -- invoke val (2nd method)
```

(c) Give assembly-level pseudocode for the call to `c.ping` at line 17. Again, assume that the address of `c` is in register `r1`, that this is the same register that should be used to pass `this`, and that you don't need to save or restore any registers.

Answer:

```
-- &c in r1
r1 += 8           -- Pingable vtable pointer
r2 = *r1          -- Pingable vtable
call *(r2+4)      -- invoke ping (1st method)
```

(d) Give assembly-level pseudocode for the body of method `Counter.ping` (again ignoring register save/restore).

Answer:

```
Counter.ping:     -- this is in r1
    r2 = *r1      -- Pingable vtable
```

```
r1 −= *r2          −− offset back to beginning of Counter object
*(r1+4)++          −− increment count
return
```

10.22 In Ruby, as in Java 8 or Scala, an interface (mix-in) can provide method code as well as signatures. (It can't provide data members; that would be multiple inheritance.) Explain why dynamic typing makes this feature more powerful than it is in the other languages.

Answer: Ruby is dynamically typed, so code obtained through a mix-in can make use of arbitrary methods. The language implementation will look these up when and if they are called at run time, and complain if they're not present. In Java 8 or Scala, interface code is limited to calling to other methods of the same interface.

IN MORE DEPTH

10.23 Suppose that class `D` inherits from classes `A`, `B`, and `C`, none of which share any common ancestor. Show how the data members and vtable(s) of `D` might be laid out in memory. Also show how to convert a reference to a `D` object into a reference to an `A`, `B`, or `C` object.

Answer:

10.24 Consider the `person_interface` and `system_user_interface` classes described in Example C-10.62. If `student` is derived from `person_interface` and `system_user_interface`, explain what happens in the following method call:

```
student s;
person *p = &s;
...
p->print_stats();
```

You may wish to use a diagram of the representation of a `student` object to illustrate the method lookups that occur and the views that are computed. You may assume an implementation akin to that of Figure C-10.10, without shared inheritance.

Answer: If we assign a reference to our `student` object into variable `p`, of type `person*`, and then call `p->print_stats`, dispatch through `*p`'s vtable (which is, of course, a prefix of our `student` vtable) will result in a call to `person_interface::print_stats` (passing, of course, a `person_interface` view of our `student` object). The code for `person_interface::print_stats` will then immediately call `print_person_stats`. Because this method is also virtual, it will dispatch through the `person_interface` portion of the `student` vtable, resulting in a call to `student::print_person_stats` (and passing a `student` view).

10.25 Given the inheritance tree of Example C-10.63, show a representation for objects of class `student_prof`. You may want to consult Figures C-10.9, C-10.10, and C-10.11.

Answer:

10.26 Given the memory layout of Figure C-10.9 and the following declarations:

```
student& sr;
system_user& ur;
```

show the code that must be generated for the assignment

```
ur = sr;
```

(Pitfall: Be sure to consider null pointers.)

Answer:

10.27 Standard C++ provides a "pointer-to-member" mechanism for classes:

```
class C {
public:
    int a;
    int b;
} c;
int C::*pm = &C::a;
    // pm points to member a of an (arbitrary) C object
...
C* p = &c;
p->*pm = 3;     // assign 3 into c.a
```

Pointers to members are also permitted for subroutine members (methods), including virtual methods. How would you implement pointers to virtual methods in the presence of C++-style multiple inheritance?

Answer:

10.28 As an alternative to using ⟨method address, `this` correction⟩ pairs in the vtable entries of a language with multiple inheritance, we could leave the entries as simple pointers, but make them point to code that updates `this` in-line, and then jumps to the beginning of the appropriate method, much as Java 8 and Scala do to implement default methods on a standard Java virtual machine. Show the sequence of instructions executed under this scheme. What factors will influence whether it runs faster or slower than the sequence shown in Example C-10.60? Which scheme will use less space? (Remember to count both code and data structure size, and consider which instructions must be replicated at every call site.)

Pursuing the replacement of data structures with executable code even further, consider an implementation in which the vtable itself consists of executable code. Show what this code would look like and, again, discuss the implications for time and space overhead.

Answer: The calling sequence looks like this:

```
r1 := &my_student      -- student view of object
r1 := r1 + d           -- gp_list_node view of object
r2 := *r1              -- address of appropriate vtable
r2 := r2[3-1]          -- scaled by sizeof(address)
call *r2
```

But the target of the call is the following:

```
r1 := r1 + c           -- this correction
goto wherever
```

This code is probably 8 instructions long (instead of 7), but 3 of the instructions are shared by all calls and the vtable entries are smaller. The new sequence uses one fewer register and makes one fewer memory access, but includes an extra jump. Perhaps most important, the thunk that adds to r1 and jumps can be omitted (and the vtable point to the regular code) whenever the `this` correction should be 0. That means we pay for multiple inheritance only when we use it.

An executable vtable entry contains the following.

```
r1 := r1 + c
goto wherever          -- 2 instructions
```

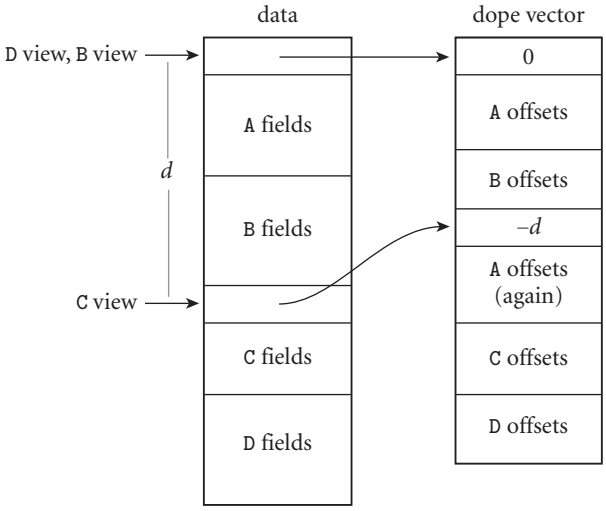
Code to call through it looks like this:

```
r1 := &my_student
r1 := r1 + d
r2 := *r1
r2 := r2 + offset      -- avoids a load
call *r2
```

Assuming the offset is small enough to fit in an immediate operand, this code sequence (again 8 instructions long) avoids one load, but at the expense of 3-word vtable entries for methods that require a `this` correction.

10.29 In Eiffel, shared inheritance is the default rather than the exception. Only renamed features are replicated. As a result, it is not possible to tell when looking at a class whether its members will be inherited replicated or shared by derived classes. Describe a uniform mechanism for looking up members inherited from base classes that will work whether they are replicated *or* shared. (Hint: Consider the use of dope vectors for records containing arrays of dynamic shape, as described in Section 8.2.2. For further details, consult the compiler text of Wilhelm and Maurer [WM95, Sec. 5.3].)

Answer: Assume, as in Sections C-10.6.2 and C-10.6.3, that `D` inherits from `B` and `C`, both of which inherit from `A`. Since `A` doesn't know which of its members will be replicated and which will be shared, the generated code cannot make any assumptions about where the members of an `A` object will lie; we need a dope vector to find them. The first field of the implementation of `D` is a pointer to the dope vector, which subsumes the usual vtable. We can get by with a single dope vector per class. The dope vector for `D` looks like this:



Every reference to a member of `B`, `C`, or `D` must indirect through this dope vector. For data members (fields), the dope vector contains an offset from the beginning of (a `D` view of) the object. For methods it contains a `this` correction as well. The size of the dope vector for `D` is the sum of the sizes of the dope vectors for `B` and `C`. `A`'s offsets appear twice, once for use by `D`s and `B`s, once for use by `C`s. Code that uses a `B` view of the object will examine only the `A` and `B` offsets that lie at the beginning of the dope vector. Code that uses a `C` view will examine only the (contiguous) `A` and `C` offsets in the middle of the dope vector. Code that uses a `D` view will examine the first set of `A` offsets, plus the `B`, `C`, and `D` offsets.

An additional set of offsets in the dope vector (`0` and $-d$ in the picture) are used to calculate a `D` view of the object, to which the field offsets can then be applied. Suppose, for example, that `c` is a reference of class `C`, and that `f` is a field of `A`. Code to load `c.f` into a register looks like this:

```
r1 := c                 -- C view of object
r2 := *r1               -- pointer to (C part of) dope vector
r3 := *r2               -- offset to beginning of D view
r1 := r1 + r3           -- D view of object
r3 := *(r2+j)           -- offset of f
r1 := r1 + r3           -- address of f
r1 := *r1               -- f
```

Here we have assumed that the offset of `f` lies at offset j within (the `C` part of) the dope vector. Code to call a method `c.m()` is similar:

```
r1 := c                 -- C view of object
r2 := *r1               -- pointer to (C part of) dope vector
```

```
r3 := *r2                  -- offset to beginning of D view
r1 := r1 + r3              -- D view of object
r3 := *(r2+k)              -- method address
r2 := *(r2+k +4)           -- this correction
r1 := r1 + r2              -- this
call *r3
```

10.30 In Figure C-10.11, consider calls to virtual methods declared in `A`, but called through a `B`, `C`, or `D` object view. We could avoid one level of indirection by appending a copy of the `A` part of the vtable to the `D/B` and `C` parts of the vtable (with suitably adjusted `this` corrections). Give calling sequences for this alternative implementation. In the worst case, how much larger may the vtable be for a class with n ancestors?

Answer:

10.31 Consider the Smalltalk implementation of Euclid's algorithm, presented at the end of Section C-10.7.1. Trace the messages involved in evaluating `4 gcd: 6`.

Answer:

11 Functional Languages

11.11 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

11.1 Is the `define` primitive of Scheme an imperative language feature? Why or why not?

Answer: Yes. The Scheme manual states: "At the top level of a program, a definition

`(define` *variable expression*`)`

has essentially the same effect as the assignment expression

`(set!` *variable expression*`)`

if *variable* is bound. If *variable* is not bound, however, then the definition will bind *variable* to a new location before performing the assignment, whereas it would be an error to perform a `set!` on an unbound variable."

11.2 It is possible to write programs in a purely functional subset of an imperative language such as C, but certain limitations of the language quickly become apparent. What features would need to be added to your favorite imperative language to make it genuinely useful as a functional language? (Hint: What does Scheme have that C lacks?)

Answer: The list is actually pretty long. Here are some possibilities:

(a) Functions as first-class objects: functional constants, function types, functions as return values, syntax to call a function returned from another function

(b) The ability to use statements as expressions, for example to embed an `if` statement (expression) in a parameter list

(c) Polymorphic functions

(d) On-the-fly creation of functions: something similar to `lambda`

(e) Literal constants of arbitrary types—constant trees, for example

(f) Garbage collection

11.3 Explain the connection between short-circuit Boolean expressions and normal-order evaluation. Why is `cond` a special form in Scheme, rather than a function?

Answer: The arguments of short-circuit operators are not evaluated unless needed to determine the result, i.e., in normal order. Scheme's `cond` is a special form because it is not supposed to evaluate later arguments if it finds an earlier one whose `car` is true.

11.4 Write a program in your favorite imperative language that has the same input and output as the Scheme program of Figure 11.1. Can you make any general observations about the usefulness of Scheme for symbolic computation, based on your experience?

Answer:

11.5 Suppose we wish to remove adjacent duplicate elements from a list (e.g., after sorting). The following Scheme function accomplishes this goal:

```
(define unique
  (lambda (L)
    (cond
      ((null? L) L)
      ((null? (cdr L)) L)
      ((eqv? (car L) (car (cdr L))) (unique (cdr L)))
      (else (cons (car L) (unique (cdr L)))))))
```

Write a similar function that uses the imperative features of Scheme to modify `L` "in place," rather than building a new list. Compare your function to the code above in terms of brevity, conceptual clarity, and speed.

Answer: Here's an imperative version:

```
(define unique2
  (lambda (L)
    (if (null? L) L
      (let ((A L) (B (cdr L)))
        (while (not (null? B))
          (if (eqv? (car A) (car B))
              (set-cdr! A (cdr B))
            (set! A (cdr A)))
          (set! B (cdr B)))
        L))))
```

The code is slightly longer. Which version is conceptually clearer would seem to depend on whether the reader tends to think in a functional or imperative style. The imperative version will be faster in most Scheme implementations.

11.6 Write tail-recursive versions of the following:

(a)
```
;; compute integer log, base 2
;; (number of bits in binary representation)
;; works only for positive integers
(define log2
  (lambda (n)
    (if (= n 1) 0 (+ 1 (log2 (quotient (+ n 1) 2))))))
```

Answer:

```
(define log2
  (lambda (n)
    (letrec
        ((helper
          (lambda (n s)
            (if (= n 1) s (helper (quotient (+ n 1) 2) (+ s 1))))))
      (helper n 0))))
```

(b)
```
;; find minimum element in a list
(define min
  (lambda (l)
    (cond
      ((null? l) '())
      ((null? (cdr l)) (car l))
      (#t (let ((a (car l))
                (b (min (cdr l))))
            (if (< b a) b a))))))
```

Answer:

```
(define min
  (lambda (l)
    (letrec
        ((helper
          (lambda (l m)
            (if (null? l) m
              (let* ((a (car l))
                     (n (if (or (null? m) (< a m)) a m)))
                (helper (cdr l) n))))))
      (helper l '()))))
```

Alternatively,

```
(define min
  (lambda (l)
    (cond
      ((null? l) '())
      ((null? (cdr l)) (car l))
      (#t (let ((a (car l))
                (b (cadr l)))
            (min (cons (if (< b a) b a) (cddr l))))))))
```

11.7 Write purely functional Scheme functions to

(a) return all *rotations* of a given list. For example, `(rotate '(a b c d e))` should return `((a b c d e) (b c d e a) (c d e a b) (d e a b c) (e a b c d))` (in some order).

Answer:

```
(define rotations
  (lambda (L)
    (letrec
        ((helper
          (lambda (Ls A B)
            (if (null? B) Ls
              (helper (append Ls (list (append B A)))
                      (append A (list (car B)))
                      (cdr B))))))
      (helper '() '() L))))
```

(b) return a list containing all elements of a given list that satisfy a given predicate. For example, `(filter (lambda (x) (< x 5)) '(3 9 5 8 2 4 7))` should return `(3 2 4)`.

Answer:

```
(define filter
  (lambda (f L)
    ; return list of those elements in L which pass through filter f
    (if (null? L) L
      (if (f (car L))
          (cons (car L) (filter f (cdr L)))
        (filter f (cdr L))))))
```

11.8 Write a purely functional Scheme function that returns a list of all permutations of a given list. For example, given `(a b c)` it should return `((a b c) (b a c) (b c a) (a c b) (c a b) (c b a))` (in some order).

Answer:

```
(define permutations
  (lambda (L)
    ; return a list of all permutations of list L
    (let
        ((insert-all
          (lambda (e Ls)
            ; insert element e into all positions in all lists in Ls
            (let
                ((insert-one
                  (lambda (L)
                    ; insert element e into all positions in list L
                    (letrec
                        ((helper
                          (lambda (L R)
                            ; insert e in all positions after L in L+R
                            (if (null? R)
                                (list (append L (list e)))
                              (append (list (append L (list e) R))
                                      (helper
                                       (append L (list (car R))) (cdr R)))))))
                      (helper '() L)))))
              (apply append (map insert-one Ls))))))
      (cond ((null? L) '())
            ((null? (cdr L)) (list L))
            (else (insert-all (car L) (permutations (cdr L))))))))
```

11.9 Modify the Scheme program of Figure 11.1 or the OCaml program of Figure 11.3 to simulate an NFA (nondeterministic finite automaton), rather than a DFA. (The distinction between these automata is described in Section 2.2.1.) Since you cannot "guess" correctly in the face of a multivalued transition function, you will need either to use explicitly coded backtracking to search for an accepting series of moves (if there is one), or keep track of *all* possible states that the machine could be in at a given point in time.

Answer:

11.10 Consider the problem of determining whether two trees have the same *fringe*: the same set of leaves in the same order, regardless of internal structure. An obvious way to solve this problem is to write a function `flatten` that takes a tree as argument and returns an ordered list of its leaves. Then we can say

```
(define same-fringe
  (lambda (T1 T2)
    (equal (flatten T1) (flatten T2))))
```

Write a straightforward version of `flatten` in Scheme. How efficient is `same-fringe` when the trees differ in their first few leaves? How would your answer differ in a language like Haskell, which uses lazy evaluation for all arguments? How hard is it to get Haskell's behavior in Scheme, using `delay` and `force`?

Answer: Here is the straightforward version of `flatten`:

```
(define flatten
  (lambda (L)
    (cond
      ((null? L) L)
      ((list? (car L)) (append (flatten (car L)) (flatten (cdr L))))
      (else (cons (car L) (flatten (cdr L)))))))
```

Comparing two flattened lists takes time proportional to the sum of the lengths of the lists, regardless of outcome. In a lazy language like Haskell, lists that differ within the first few elements would compare in constant time.

11.11 In Example 11.59 we showed how to implement interactive I/O in terms of the lazy evaluation of streams. Unfortunately, our code would not work as written, because Scheme uses applicative-order evaluation. We can make it work, however, with calls to `delay` and `force`.

Suppose we define `input` to be a function that returns an "istream"—a promise that when forced will yield a pair, the `cdr` of which is an istream:

```
(define input (lambda () (delay (cons (read) (input)))))
```

Now we can define the driver to expect an "ostream"—an empty list or a pair, the `cdr` of which is an ostream:

```
(define driver
  (lambda (s)
    (if (null? s) '()
        (display (car s))
        (driver (force (cdr s))))))
```

Note the use of `force`.

Show how to write the function `squares` so that it takes an istream as argument and returns an ostream. You should then be able to type `(driver (squares (input)))` and see appropriate behavior.

Answer:

```
(define squares
  (lambda (s)
    (cons "please enter a number\n"
          (delay
           (let* ((p (force s))
                  (n (car p)))
             (if (eof-object? n) '()
                 (cons (* n n)
                       (delay
                         (cons #\newline
                               (delay (squares (cdr p))))))))))))
```

11.12 Write new versions of `cons`, `car`, and `cdr` that operate on streams. Using them, rewrite the code of the previous exercise to eliminate the calls to `delay` and `force`. Note that the stream version of `cons` will need to avoid evaluating its second argument; you will need to learn how to define macros (derived special forms) in Scheme.

Answer:

11.13 Write the standard quicksort algorithm in Scheme, without using any imperative language features. Be careful to avoid the trivial update problem; your code should run in expected time $n \log n$.

Rewrite your code using arrays (you will probably need to consult a Scheme manual for further information). Compare the running time and space requirements of your two sorts.

Answer: Here is a solution for lists of numbers. It is simplistic in that it partitions on the first element of the list (so it will take quadratic time on an already sorted list) and it doesn't revert to an always-quadratic but low-constant-overhead sort (e.g., insertion sort) on small lists.

```
(define sort
  (lambda (L)
    (letrec ((partition
              (lambda (e L A B)
                (if (null? L) (cons A B)
                    (let ((c (car L)))
                      (if (< c e)
                          (partition e (cdr L) (cons c A) B)
                          (partition e (cdr L) A (cons c B))))))))
      (cond
        ((null? L) L)
        ((null? (cdr L)) L)
        (else (let* ((AB (partition (car L) (cdr L) '() '()))
                     (A (car AB))
                     (B (cdr AB)))
                (append (sort A)
                        (list (car L))
                        (sort B))))))))
```

11.14 Write `insert` and `find` routines that manipulate binary search trees in Scheme (consult an algorithms text if you need more information). Explain why the trivial update problem does *not* impact the asymptotic performance of `insert`.

Answer:

11.15 Write an LL(1) parser generator in purely functional Scheme. If you consult Figure 2.24, remember that you will need to use tail recursion in place of iteration. Assume that the input CFG consists of a list of lists, one per nonterminal in the grammar. The first element of each sublist should be the nonterminal; the remaining elements should be the right-hand sides of the productions for which that nonterminal is the left-hand side. You may assume that the sublist for the start symbol will be the first one in the list. If we use quoted strings to represent grammar symbols, the calculator grammar of Figure 2.16 would look like this:

```
'(("program"  ("stmt_list" "$$"))
  ("stmt_list" ("stmt" "stmt_list") ())
  ("stmt"  ("id" ":=" "expr") ("read" "id") ("write" "expr"))
  ("expr"  ("term" "term_tail"))
  ("term"  ("factor" "factor_tail"))
  ("term_tail" ("add_op" "term" "term_tail") ())
  ("factor_tail" ("mult_op" "factor" "FT") ())
  ("add_op" ("+") ("-"))
  ("mult_op" ("*") ("/"))
  ("factor"  ("id") ("number") ("(" "expr" ")")))
```

Your output should be a parse table that has this same format, except that every right-hand side is replaced by a *pair* (a 2-element list) whose first element is the predict set for the corresponding production, and whose second element is the right-hand side. For the calculator grammar, the table looks like this:

```
(("program" (("$$" "id" "read" "write") ("stmt_list" "$$")))
 ("stmt_list"
  (("id" "read" "write") ("stmt" "stmt_list"))
  (("$$") ()))
 ("stmt"
  (("id") ("id" ":=" "expr"))
  (("read") ("read" "id"))
  (("write") ("write" "expr")))
 ("expr" ((("(" "id" "number") ("term" "term_tail")))
 ("term" ((("(" "id" "number") ("factor" "factor_tail")))
 ("term_tail"
  (("+" "-") ("add_op" "term" "term_tail"))
  (("$$" ")" "id" "read" "write") ()))
 ("factor_tail"
  (("*" "/") ("mult_op" "factor" "factor_tail"))
  (("$$" ")" "+" "-" "id" "read" "write") ()))
 ("add_op" (("+") ("+")) (("-") ("-")))
```

```
    ("mult_op" (("*") ("*")) (("/") ("/")))
    ("factor"
     (("id") ("id"))
     (("number") ("number"))
     (("(") ("(" "expr" ")"))))
```

(Hint: You may want to define a `right_context` function that takes a nonterminal B as argument and returns a list of all pairs (A, β), where A is a nonterminal and β is a list of symbols, such that for some potentially different list of symbols α, $A \longrightarrow \alpha\ B\ \beta$. This function is useful for computing FOLLOW sets. You may also want to build a tail-recursive function that recomputes FIRST and FOLLOW sets until they converge. You will find it easier if you do not include ϵ in either set, but rather keep a separate estimate, for each nonterminal, of whether it may generate ϵ.)

Answer:

```
(define sort
  (lambda (L)
    ; Use string comparison to quicksort list.
    (letrec ((partition
              (lambda (e L A B)
                (if (null? L) (cons A B)
                    (let ((c (car L)))
                      (if (string<? c e)
                          (partition e (cdr L) (cons c A) B)
                          (partition e (cdr L) A (cons c B))))))))
      (cond
        ((null? L) L)
        ((null? (cdr L)) L)
        (else (let* ((AB (partition (car L) (cdr L) '() '()))
                     (A (car AB))
                     (B (cdr AB)))
                (append (sort A)
                        (list (car L))
                        (sort B))))))))

(define unique
  (lambda (L)
    ; Return list in which adjacent equal elements have been combined.
    (cond
      ((null? L) L)
      ((null? (cdr L)) L)
      ((equal? (car L) (cadr L)) (unique (cdr L)))
      (else (cons (car L) (unique (cdr L)))))))

(define unique-sort (lambda (L) (unique (sort L))))
    ; Sort (using string-ified elements) and remove duplicates.
```

```
(define flatten
  (lambda (L)
    ; Return left-to-right fringe of tree as list.
    (cond
      ((null? L) L)
      ((list? (car L)) (append (flatten (car L)) (flatten (cdr L))))
      (else (cons (car L) (flatten (cdr L)))))))

(define start-symbol (lambda (grammar) (caar grammar)))

(define last
  (lambda (L)
    ; Return last element of list.
    (cond
      ((null? L) '())
      ((null? (cdr L)) (car L))
      (else (last (cdr L))))))

(define non-terminals (lambda (grammar) (map car grammar)))
    ; Return list of all non-terminals in grammar.

(define gsymbols (lambda (grammar) (unique-sort (flatten grammar))))
    ; Return list of all symbols in grammar (no duplicates).

(define terminals
  (lambda (grammar)
    ; Return list of all terminals in grammar.
    (apply append
           (map (lambda (x) (if (member x (non-terminals grammar))
                                '()
                                (list x)))
                (gsymbols grammar)))))

(define productions
  (lambda (grammar)
    ; Return list of all productions in grammar.
    ; Each is represented as a (lhs rhs) pair, where rhs is a list.
    (apply append
           (map (lambda (prods)
                  (map (lambda (rhs)
                         (list (car prods) rhs))
                       (cdr prods)))
                grammar))))
```

```
(define terminal?
  (lambda (x grammar)
    ; Is x a terminal in grammar?
    (and (member x (gsymbols grammar))
         (not (member x (non-terminals grammar))))))

(define union (lambda sets (unique-sort (apply append sets))))
    ; Set union, where arguments are (flat) lists

(define right-context
  (lambda (B grammar)
    ; Return a list of pairs.
    ; Each pair consists of a symbol A and a list of symbols beta
    ; such that for some alpha, A -> alpha B beta.
    (apply append
           (map
            (lambda (prod)
              (letrec
                  ((helper
                    (lambda (subtotal rhs)
                      (let ((suffix (member B rhs)))
                        (if suffix
                            (helper (cons (cons (car prod)
                                                (list (cdr suffix)))
                                          subtotal)
                                    (cdr suffix))
                            subtotal)))))
                 (helper '() (cadr prod))))
            (productions grammar)))))

;; Much of the following employs a "knowledge" structure.
;; It's a list of 4-tuples, one for each non-terminal,
;;    in the same order those non-terminals appear in the grammar
;;    (the order is important).
;;
;; The fields of the 4-tuple are:
;;   car     non-terminal A [not needed computationally,
;;           but included for readability of output]
;;   cadr    Boolean: do we currently think A can -->* epsilon
;;   caddr   (current guess at) FIRST(A) - {epsilon}
;;   cadddr  (current guess at) FOLLOW(A) - {epsilon}
```

```
(define initial-knowledge
  (lambda (grammar)
    ; Return knowledge structure with empty FIRST and FOLLOW sets
    ; and false gen-epsilon estimate for all symbols.
    (map (lambda (A) (list A #f '() '()))
         (non-terminals grammar))))

(define symbol-knowledge (lambda (A knowledge) (assoc A knowledge)))
    ; Return knowledge vector for A.

(define generates-epsilon?
  (lambda (w knowledge grammar)
    ; Can w generate epsilon based on current estimates?
    ; if w is a terminal, no
    ; if w is a non-terminal, look it up
    ; if w is an empty list, yes
    ; if w is a non-empty list, "iterate" over elements
    (cond
      ((terminal? w grammar) #f)
      ((list? w)
       (or (null? w)
           (and (generates-epsilon? (car w) knowledge grammar)
                (generates-epsilon? (cdr w) knowledge grammar))))
      (else (cadr (symbol-knowledge w knowledge))))))

(define first
  (lambda (w knowledge grammar)
    ; Return FIRST(w) - {epsilon}, based on current estimates.
    ; if w is a terminal, return (w)
    ; if w is a non-terminal, look it up
    ; if w is an empty list, return ()  [empty set]
    ; if w is a non-empty list, "iterate" over elements
    (cond
      ((terminal? w grammar) (list w))
      ((null? w) '())
      ((list? w)
       (union (first (car w) knowledge grammar)
              (if (generates-epsilon? (car w) knowledge grammar)
                  (if (null? (cdr w))
                      '()
                      (first (cdr w) knowledge grammar))
                  '())))
      (else (caddr (symbol-knowledge w knowledge))))))
```

```
(define follow
  (lambda (A knowledge)
    ; Return FOLLOW(A) - {epsilon}, based on current estimates.
    ; Simply look it up.  This is simpler than the preceding
    ; two routines because its argument is assumed to be a single
    ; non-terminal -- never a list.
    (cadddr (symbol-knowledge A knowledge))))

(define get-knowledge
  (lambda (grammar)
    ; Return knowledge structure for grammar.
    ; Start with (initial-knowledge grammar) and "iterate", until
    ; the structure doesn't change.
    ; Uses (right-context B grammar), for all non-terminals B,
    ; to help compute follow sets.
    (letrec
        ((nts (non-terminals grammar))
         (right-contexts
          (map (lambda (s) (right-context s grammar)) nts))
         (helper
          (lambda (knowledge)
            (let*
                ((update
                  (lambda (old-symbol-knowledge
                           symbol-productions
                           symbol-right-context)
                    (let*
                        ((local-first
                          (lambda (s) (first s knowledge grammar)))
                         (local-gen-ep?
                          (lambda (s)
                            (generates-epsilon? s knowledge grammar)))
                         (filtered-follow
                          (lambda (p) (if (local-gen-ep? (cadr p))
                                          (follow (car p) knowledge)
                                          '()))))
                      (list
                       (car old-symbol-knowledge) ; non-terminal itself
                       (or (cadr old-symbol-knowledge)
                           (eval (cons 'or
                                       (map local-gen-ep?
                                            (cdr symbol-productions)))))
                       (union (caddr old-symbol-knowledge) ; previous estimate
                              (apply union (map local-first
                                                (cdr symbol-productions))))
```

```
                        (union (cadddr old-symbol-knowledge) ; previous estimate
                               (apply union (map local-first
                                                 (map cadr symbol-right-context)))
                               (apply union (map filtered-follow
                                                 symbol-right-context)))))))
                   (new-knowledge (map update knowledge grammar right-contexts)))
              (if (equal? new-knowledge knowledge) ; recursive comparison
                  knowledge
                  (helper new-knowledge))))))
      (helper (initial-knowledge grammar)))))

(define parse-table
  (lambda (grammar)
    ; Return parse table for grammar.
    ; Table looks like the grammar, except that each RHS is replaced with a
    ; (predict-set RHS) pair.
    ; My version uses the get-knowledge routine above.
    (let ((knowledge (get-knowledge grammar)))
      (map
       (lambda (prods)
         (let ((lhs (car prods))
               (rhss (cdr prods)))
           (cons lhs
                 (map
                  (lambda (rhs)
                    (cons (union
                           (first rhs knowledge grammar)
                           (if (generates-epsilon? rhs knowledge grammar)
                               (follow lhs knowledge)
                               '()))
                          (list rhs)))
                  rhss))))
       grammar))))

(define lookup
  (lambda (nt t parse-tab)
    ; Double-index to find prediction for non-terminal nt and terminal t.
    ; Return #f if not found.
    (letrec ((helper
              (lambda (L)
                (cond
                  ((null? L) #f)
                  ((member t (caar L)) (cadar L))
                  (else (helper (cdr L)))))))
      (helper (cdr (assoc nt parse-tab))))))
```

```
(define display-list
  (lambda (L)
    ; Print list to standard output.
    ; Yes, this is imperative.
    (if (not (null? L))
        (begin (display (string-append " " (car L)))
               (display-list (cdr L))))))

(define parse
  (lambda (grammar input)
    ; Parse input according to grammar.
    ; Print predictions and matches (imperatively) along the way.
    ; You can also print the stack and remaining input at each step,
    ; if you want: simply uncomment the ';;;' line below
    ; Return #t if the input is in the language; #f if it's not.
    (letrec
        ((die (lambda (s) (begin (display "syntax error: ")
                                 (display s) (newline) #f)))
         (parse-tab (parse-table grammar))
         (helper
          (lambda (stack input)
            (cond
              ((null? stack)
               (or (null? input)
                   (die "extra input beyond end of program")))
              ((null? input)
               (die (string-append "end of input when "
                                   (car stack) " is expected")))
              ((terminal? (car stack) grammar)
               (if (equal? (car stack) (car input))
                   (begin
                     (display (string-append "   match " (car input)))
                     (newline)
                     (helper (cdr stack) (cdr input)))
                   (die (string-append "expected " (car stack)
                                       "; saw " (car input)))))
```

```
                (else ; non-terminal
                 (let ((rhs (lookup (car stack) (car input) parse-tab)))
                   (if rhs
                       (begin
                         (display (string-append "   predict "
                                                 (car stack) " ->"))
                         (display-list rhs)
                         (newline)
                         (helper (append rhs (cdr stack)) input))
                       (die (string-append "no prediction for "
                                           (car stack)
                                           " when seeing "
                                           (car input)))))))))
      (helper (list (start-symbol grammar)) input))))
```

11.16 Write an equality operator (call it `=/`) that works correctly on the `yearday` type of Example 11.38. (You may need to look up the rules that govern the occurrence of leap years.)

Answer:

```
type yearday = YMD of int * int * int | YD of int * int;;

let leap yr = (yr mod 4 = 0) && not (yr mod 100 = 0);;

let day_of_year = function
| (YD (y, d))        -> d
| (YMD (y, m, d)) ->
    let month_lengths = [| 31; 28; 31; 30; 31; 30; 31; 31; 30; 31; 30; 31 |] in
    let rec total_days m s =
      if m < 2 then s
      else total_days (m-1) (s + month_lengths.(m-2)) in
    let nld = total_days m d in
    if (leap y) && (m > 2) then nld + 1 else nld;;

let (=/) (date1:yearday) (date2:yearday) =
  match (date1, date2) with
  | (YMD (y1, m1, d1), YMD (y2, m2, d2)) -> (y1 = y2) && (m1 = m2) && (d1 = d2)
  | (YMD (y1, m2, d1), YD (y2, d2))      -> (y1 = y2) && ((day_of_year date1) = d2)
  | (YD (y1, d1), YMD (y2, m2, d2))      -> (y1 = y2) && (d1 = (day_of_year date2))
  | (YD (y1, d1), YD (y2, d2))           -> (y1 = y2) && (d1 = d2);;
```

11.17 Create addition and subtraction functions for the celsius and fahrenheit temperature types introduced in Sidebar 11.3. To allow the two scales to be mixed, you should also define conversion functions `ct_of_ft : fahrenheit_temp -> celsius_temp` and `ft_of_ct : celsius_temp -> fahrenheit_temp`. Your conversions should round to the nearest degree (half degrees round up).

Answer:

```
let round x = int_of_float (floor (x +. 0.5));;
let ct_of_ft (FT f) = CT (round (float_of_int (f - 32) *. 5. /. 9.));;
let ft_of_ct (CT c) = FT (round (float_of_int c /. 9. *. 5.) + 32);;

let cplus (CT a) (CT b) = CT(a+b);;
let fplus (FT a) (FT b) = FT(a+b);;
```

11.18 We can use encapsulation within functions to delay evaluation in OCaml:

```
type 'a delayed_list =
  Pair of 'a * 'a delayed_list
| Promise of (unit -> 'a * 'a delayed_list);;

let head = function
| Pair (h, r) -> h
| Promise (f) -> let (a, b) = f() in a;;

let rest = function
| Pair (h, r) -> r
| Promise (f) -> let (a, b) = f() in b;;
```

Now given

```
let rec next_int n = (n, Promise (fun() -> next_int (n + 1)));;
let naturals = Promise (fun() -> next_int (1));
```

we have

```
head naturals                 ⟹ 1
head (rest naturals)          ⟹ 2
head (rest (rest naturals))   ⟹ 3
...
```

The delayed list `naturals` is effectively of unlimited length. It will be computed out only as far as actually needed. If a value is needed more than once, however, it will be recomputed every time. Show how to use pointers and assignment (Example 8.42) to memoize the values of a `delayed_list`, so that elements are computed only once.

Answer:

```
type 'a memoized_list =
  MPair    of ('a * 'a memoized_list ref)
| MPromise of (unit -> 'a * 'a memoized_list ref);;

let mhead p = match !p with
| MPair (a, b) -> a
| MPromise (f) -> let (a, b) as n = print_char '*'; f() in p := MPair n; a;;

let mrest p = match !p with
| MPair (a, b) -> b
| MPromise (f) -> let (a, b) as n = print_char '*'; f() in p := MPair n; b;;

let rec next_m_int n = (n, ref (MPromise (fun() -> next_m_int (n + 1))));;
let mnaturals = ref (MPromise (fun() -> next_m_int 1));;
```

11.19 Write an OCaml version of Example 11.67. Alternatively (or in addition), solve Exercises 11.5, 11.7, 11.8, 11.10, 11.13, 11.14, or 11.15 in OCaml.

Answer:

IN MORE DEPTH

11.20 In Figure C-11.6 we evaluated our expression in normal order. Did we really have any choice? What would happen if we tried to use applicative order?

Answer:

11.21 Prove that for any lambda expression f, if the normal-order evaluation of $\mathbf{Y}f$ terminates, where $\mathbf{Y}$ is the fixed-point combinator $\lambda h.(\lambda x.h(x\,x))\,(\lambda x.h(x\,x))$, then $f(\mathbf{Y}f)$ and $\mathbf{Y}f$ will reduce to the same simplest form.

Answer:

11.22 Given the definition of structures (lists) in Section C-11.7.3, what happens if we apply car or cdr to nil? How might you introduce the notion of "type error" into lambda calculus?

Answer:

11.23 Let

$$\text{zero} \equiv \lambda x.x$$

$$\text{succ} \equiv \lambda n.(\lambda s.(s\ \text{select\_second})\ n)$$

where $\text{select\_second} \equiv \lambda x.\lambda y.y$. Now let

$$\text{one} \equiv \text{succ zero}$$

$$\text{two} \equiv \text{succ one}$$

Show that

$$\text{one select\_second} = \text{zero}$$
$$\text{two select\_second select\_second} = \text{zero}$$

In general, show that

$$\text{succ}^n \text{ zero select\_second}^n = \text{zero}$$

Use this result to define a predecessor function pred. You may ignore the issue of the predecessor of zero.

Note that our definitions of T and F allow us to check whether a number is equal to zero:

$$\text{iszero} \equiv \lambda n.(n \text{ select\_first})$$

Using succ, pred, iszero, and if, show how to define plus and times recursively. These definitions could of course be made nonrecursive by means of beta abstraction and **Y**.

Answer:

12 Logic Languages

12.7 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

12.1 Starting with the clauses at the beginning of Example 12.17, use resolution (as illustrated in Example 12.3) to show, in two different ways, that there is a path from `a` to `e`.

Answer:

12.2 Solve Exercise 6.22 in Prolog.

Answer:

12.3 Consider the Prolog `gcd` program in Figure 1.2. Does this program work "backward" as well as forward? (Given integers d and n, can you use it to generate a sequence of integers m such that $\texttt{gcd}(n, m) = d$?) Explain your answer.

Answer: No, it only works forward. The use of `is` in the second and third rules requires that `A` and `B` both be instantiated.

12.4 In the spirit of Example 11.20, write a Prolog program that exploits backtracking to simulate the execution of a *non*deterministic finite automaton.

Answer:

12.5 Show that resolution is commutative and associative. Specifically, if A, B, and C are Horn clauses, show that $(A \oplus B) = (B \oplus A)$ and that $((A \oplus B) \oplus C) = (A \oplus (B \oplus C))$, where $\oplus$ indicates resolution. Be sure to think about what happens to variables that are instantiated as a result of unification.

Answer:

12.6 In Example 12.8, the query `?- classmates(jane_doe, X)` will succeed three times: twice with `X` = `jane_doe`, and once with `X` = `ajit_chandra`. Show how to modify the `classmates(X, Y)` rule so that a student is not considered a classmate of him or herself.

Answer: `classmates(X, Y) :- takes(X, Z), takes(Y, Z), \+(X = Y)`. This can also be written `classmates(X, Y) :- takes(X, Z), takes(Y, Z), X \= Y`.

12.7 Modify Example 12.17 so that the goal `path(X, Y)`, for arbitrary already-instantiated `X` and `Y`, will succeed no more than once, even if there are multiple paths from `X` to `Y`.

Answer:

12.8 Using only `\+` (no cuts), modify the tic-tac-toe example of Section 12.2.5 so it will generate only one candidate move from a given board position. How does your solution compare to the cut-based one (Example 12.22)?

Answer:

12.9 Prove the claim, made in Example 12.19, that there is no winning strategy in tic-tac-toe—that either player can force a draw.

Answer:

12.10 Prove that the tic-tac-toe strategy of Example 12.19 is optimal (wins against an imperfect opponent whenever possible, draws otherwise), or give a counterexample.

Answer:

12.11 Starting with the tic-tac-toe program of Figure 12.4, draw a directed acyclic graph in which every clause is a node and an arc from `A` to `B` indicates that it is important, either for correctness or efficiency, that `A` come before `B` in the program. (Do not draw any other arcs.) Any topological sort of your graph should constitute an equally efficient version of the program. (Is the existing program one of them?)

Answer:

12.12 Write Prolog rules to define a version of the `member` predicate that will generate all members of a list during backtracking, but without generating duplicates. Note that the cut and `\+` based versions of Example 12.20 will not suffice; when asked to look for an uninstantiated member, they find only the head of the list.

Answer:

12.13 Use the `clause` predicate of Prolog to implement the `call` predicate (pretend that it isn't built in). You needn't implement all of the built-in predicates of Prolog; in particular, you may ignore the various imperative control-flow mechanisms and database manipulators. Extend your code by making the database an explicit argument to `call`, effectively producing a metacircular interpreter.

Answer:

12.14 Use the `clause` predicate of Prolog to write a predicate `call_bfs` that attempts to satisfy goals breadth-first. (Hint: You will want to keep a queue of yet-to-be-pursued subgoals, each of which is represented by a stack that captures backtracking alternatives.)

Answer:

12.15 Write a (list-based) *insertion sort* algorithm in Prolog. Here's what it looks like in C, using arrays:

```
void insertion_sort(int A[], int N)
{
    int i, j, t;
    for (i = 1; i < N; i++) {
        t = A[i];
        for (j = i; j > 0; j--) {
            if (t >= A[j-1]) break;
            A[j] = A[j-1];
        }
        A[j] = t;
    }
}
```

Answer:

12.16 Quicksort works well for large lists, but has higher overhead than insertion sort for short lists. Write a sort algorithm in Prolog that uses quicksort initially, but switches to insertion sort (as defined in the previous exercise) for sublists of 15 or fewer elements. (Hint: You can count the number of elements during the `partition` operation.)

Answer:

12.17 Write a Prolog sorting routine that is guaranteed to take $O(n \log n)$ time in the worst case. (Hint: Try *merge sort*; a description can be found in almost any algorithms or data structures text.)

Answer:

12.18 Consider the following interaction with a Prolog interpreter:

```
?- Y = X, X = foo(X).
Y = foo(foo(foo(foo(foo(foo(foo(foo(foo(foo(foo(
foo(foo(foo(foo(foo(foo(foo(foo(foo(foo(foo(foo(
foo(foo(foo(foo(foo(foo(foo(foo(foo(foo(foo(foo(
foo(foo(foo(foo(foo(foo(foo(foo(foo(foo(foo(foo(
foo(foo(foo(foo(foo(foo(...
```

What is going on here? Why does the interpreter fall into an infinite loop? Can you think of any circumstances (presumably not requiring output) in which a structure like this one would be useful? If not, can you suggest how a Prolog interpreter might implement checks to forbid its creation? How expensive would those checks be? Would the cost in your opinion be justified?

Answer:

IN MORE DEPTH

12.19 Restate the following Prolog rule in predicate calculus, using appropriate quantifiers:

```
sibling(X, Y) :- mother(M, X), mother(M, Y),
                 father(F, X), father(F, Y).
```

Answer: There are several possible formulas. Here's one:

$$\forall X, \forall Y[\text{sibling}(X, Y) \leftarrow (\exists M, \exists F[\text{mother}(M, X) \wedge \text{mother}(M, Y) \wedge \text{father}(F, X) \wedge \text{father}(F, Y)])].$$

12.20 Consider the following statement in predicate calculus:

$$\text{empty\_class}(C) \leftarrow \neg\exists X[\text{takes}(X, C)]$$

(a) Translate this statement to clausal form.

(b) Can you translate the statement into Prolog? Does it make a difference whether you're allowed to use `\+`?

(c) How about the following:

$$\text{takes\_everything}(X) \leftarrow \forall C[\text{takes}(X, C)]$$

Can this be expressed in Prolog?

Answer: The first statement has the following clausal form:

$$\text{empty\_class}(C) \vee \text{takes}(\text{distinguished\_student\_in}(C), C)$$

This does not translate directly into Prolog. If we're allowed `\+`, however, we can write

```
empty_class(C) :- \+(takes(S, C)).
```

We can translate the second statement into clausal form as follows:

$$\text{takes\_everything}(X) \vee \neg\forall C[\text{takes}(X, C)]$$
$$\text{takes\_everything}(X) \vee \exists C[\neg\text{takes}(X, C)]$$
$$\text{takes\_everything}(X) \vee \neg\text{takes}(X, \text{missing\_class\_of}(X))$$

There does not appear to be any way to express this in Prolog.

12.21 Consider the seemingly contradictory statement

$$\neg \text{foo(X)} \rightarrow \text{foo(X)}$$

Convert this statement to clausal form, and then translate into Prolog. Explain what will happen if you ask

```
?- foo(bar).
```

Now consider the straightforward translation, without the intermediate conversion to clausal form:

```
foo(X) :- \+(foo(X)).
```

Now explain what will happen if you ask

```
?- foo(bar).
```

Answer:

13 Concurrency

13.8 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

13.1 Give an example of a "benign" race condition—one whose outcome affects program behavior, but not correctness.

Answer: (One of many possible.) In a "bag of tasks" system, it doesn't generally matter which thread gets to the "bag" first: any task can be performed by any thread.

13.2 We have defined the *ready list* of a thread package to contain all threads that are runnable but not running, with a separate variable to identify the currently running thread. Could we just as easily have defined the ready list to contain *all* runnable threads, with the understanding that the one at the head of the list is running? (Hint: Think about multiprocessors.)

Answer: The proposed change works fine on a uniprocessor, but on a multiprocessor we need multiple "current running thread"s.

13.3 Imagine you are writing the code to manage a hash table that will be shared among several concurrent threads. Assume that operations on the table need to be atomic. You could use a single mutual exclusion lock to protect the entire table, or you could devise a scheme with one lock per hash-table bucket. Which approach is likely to work better, under what circumstances? Why?

Answer: A single lock is easy to code and will work quite well so long as the number of threads contending for simultaneous access is not large enough to make the table a bottleneck. Multiple locks will permit a higher degree of concurrency, but they incur higher overhead in the no-contention case, since several of them may have to be acquired and released to perform a single operation. Moreover, it may be tricky to write the multiple-lock version correctly: if for example one process tries to lock buckets *A* and *B*, in that order, while another tries to lock *B* and *A*, in *that* order, deadlock may result.

13.4 The typical spin lock holds only one bit of data, but requires a full word of storage, because only full words can be read, modified, and written atomically in hardware. Consider, however, the hash table of the previous exercise. If we choose to employ a separate lock for each bucket of the table, explain how to implement a "two-level" locking scheme that couples a conventional spin lock for the table as a whole with a *single bit* of locking information for each bucket. Explain why such a scheme might be desirable, particularly in a table with external chaining.

Answer:

13.5 Drawing inspiration from Examples 13.29 and 13.30, design a nonblocking linked-list implementation of a stack using `compare_and_swap`. (When `CAS` was first introduced, on the IBM 370 architecture, this algorithm was one of the driving applications [Tre86].)

Answer: Assume we are working in a language with automatic garbage collection and a reference model of variables. The following pseudocode should do the trick:

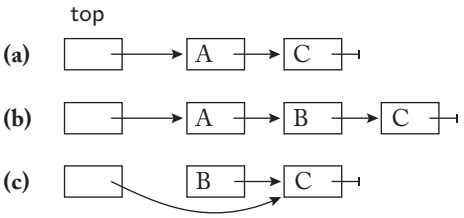
```
class node<T>
    val : T
    next : node<T>

class stack<T>
    top : node<T> := null

    method push(v : T)
        var o := top
        var n := new node<T>(v, o)
        while !CAS(top, o, n)
            o := top
            n.next := o

    method pop() : T
        o : node<T>
        n : node<T>
        repeat
            o := top
            n := o.next
        until CAS(top, o, n)
        return o.val
```

13.6 Building on the previous exercise, suppose that stack nodes are dynamically allocated. If we read a pointer and then are delayed (e.g., due to preemption), the node

to which the pointer refers may be reclaimed and then reallocated for a different purpose. A subsequent `compare-and-swap` may then succeed when logically it should not. This issue is known as the *ABA problem*.

Give a concrete example—an interleaving of operations in two or more threads—where the ABA problem may result in incorrect behavior for your stack. Explain why this behavior cannot occur in systems with automatic garbage collection. Suggest what might be done to avoid it in systems with manual storage management.

Answer: Suppose, in a language without automatic garbage collection, that we augment a solution to the previous exercise to manually delete the old node from the stack before returning from pop. Consider the following scenario:



In (a), our stack includes nodes A and C. Suppose that thread 1 begins to execute pop(), and has reached but not yet executed its CAS operation. Now suppose that thread 2 executes a (complete) pop() operation, followed by push(y) and push(x). Suppose further that the second push, when it allocates a node to hold x, happens to obtain the same heap block that was deleted by the initial pop in thread 2. This will leave the stack as shown in (b). If thread 1 now continues, its CAS will succeed, leaving the stack in the broken state shown in (c).

With automatic garbage collection, the existence of a reference to node A in thread 1 will prevent the collector from reclaiming—and thread 2 from reusing—the block.

The standard solution in languages with manual reclamation is to make top a ⟨node<T>, int⟩ pair, where the int simply counts the number of successful pop operations:

```
class node<T>
    val : T
    next : node<T>

type counted_ptr<U> = ⟨U, int⟩

class stack<T>
    top : counted_ptr<node<T>> := ⟨null, 0⟩

    method push(v : T)
        ⟨o, c⟩ := top
        var n := new node<T>(v, o)
        while !CAS(top, ⟨o, c⟩, ⟨n, c⟩)
            ⟨o, c⟩ := top
            n.next := o

    method pop() : T
        o : node<T>
        n : node<T>
```

```
repeat
    ⟨o, c⟩ := top
    n := o.next
until CAS(top, ⟨o, c⟩, ⟨n, c+1⟩)
return o.val
```

13.7 We noted in Section 13.3.2 that several processors, including the ARM, MIPS, and Power, provide an alternative to `compare_and_swap` (`CAS`) known as `load_linked/store_conditional` (`LL/SC`). A `load_linked` instruction loads a memory location into a register and stores certain bookkeeping information into hidden processor registers. A `store_conditional` instruction stores the register back into the memory location, but only if the location has not been modified by any other processor since the `load_linked` was executed. Like `compare_and_swap`, `store_conditional` returns an indication of whether it succeeded or not.

(a) Rewrite the code sequence of Example 13.29 using `LL/SC`.

Answer:

```
start:
    r1 := LL(x)
    r2 := foo(r1)          -- probably a multi-instruction sequence
    r2 := SC(x, r2)        -- replace x if it hasn't changed
    if !r2 goto start
```

(b) On most machines, an `SC` instruction can fail for any of several "spurious" reasons, including a page fault, a cache miss, or the occurrence of an interrupt in the time since the matching `LL`. What steps must a programmer take to make sure that algorithms work correctly in the face of such failures?

Answer: The code must not deterministically induce a spurious failure between the `LL` and the `SC`. On some machines, this means the code must not perform any load or store instructions, because they might accidentally map to the same location in the cache as the one occupied by the `LL` operand. In addition, the code must loop back in the event of failure, even in algorithms where failure can be caused *only* by spurious conditions.

(c) Discuss the relative advantages of `LL/SC` and `CAS`. Consider how they might be implemented on a cache-coherent multiprocessor. Are there situations in which one would work but the other would not? (Hints: Consider algorithms in which a thread may need to touch more than one memory location. Also consider algorithms in which the contents of a memory location might be changed and then restored, as in the previous exercise.)

Answer: (1) `LL/SC` is immune to the ABA problem (Exercise 13.6): if a location is overwritten with the same value it previously held, a subsequent `SC` will fail. (2) A `CAS`-based algorithm can safely perform an arbitrarily complex computation between loading a value and trying to update it with `CAS`. Algorithms based on `LL/SC` may be limited in the number of instructions and/or the number of memory accesses than can safely occur in-between. From a purely theoretical point of view, the possibility of spurious failures also means that `LL/SC`-based algorithms are generally not formally nonblocking (there is no a priori bound

on the number of instructions required for forward progress). (3) If an algorithm *is* allowed to read an unrelated location between `LL` and `SC`, there are times in which `LL`/`SC` (but not `CAS`) can be used to support a limited form of "compare-two-locations-and-swap-one" operation. For an example, see the poster paper by Marathe and Scott at *PODC* 2005.

13.8 Starting with the test-and-`test_and_set` lock of Figure 13.8, implement busy-wait code that will allow readers to access a data structure concurrently. Writers will still need to lock out both readers and other writers. You may use any reasonable atomic instruction(s) (e.g., `LL`/`SC`). Consider the issue of fairness. In particular, if there are *always* readers interested in accessing the data structure, your algorithm should ensure that writers are not locked out forever.

Answer:

13.9 Assuming the Java memory model,

(a) Explain why it suffices in Figure 13.11 to label X and Y as `volatile`.

Answer: Labeling a variable `volatile` in Java ensures that a read of it is ordered before any subsequent operations in the same thread, and a write of it is ordered after any previous operations in the same thread. It also ensures a global total ordering with respect to all reads and writes of all `volatile` variables. These rules force the two reads in C to occur in program order; likewise the two reads in D. They also C and D to agree on the order in which X and Y were written.

(b) In that same figure, without `volatile` labels, explain why it suffices to enclose each thread's code in a `synchronized` block for some common shared object O, but it does *not* suffice to enclose only the pairs of reads in C and D.

Answer: Even if X and Y are not `volatile`, `synchronized` blocks in C and D ensure that one pair of reads occurs before the release of O's lock, which occurs before the acquire of O's lock in the other thread, which in turn occurs before that other thread's pair of reads. If the writes in A and B are also enclosed in `synchronized` blocks, then we know that they will be ordered with respect to the pairs of reads, and C and D will agree on the order of the writes. Without the `synchronized` blocks in A and B, each read races with the write of the corresponding variable, and the Java memory model allows the writes to be seen or not seen, nondeterministically.

(c) Explain why it is sufficient, in Example 13.31, to label both inspected and X as `volatile`, but not to label only one.

Answer: Since there is a total order on all reads and writes of all `volatile` variables, a loop becomes impossible if both variables are labeled. But suppose only is labeled—without loss of generality, say it's inspected. If B sees inspected = false, then its read is unambiguously ordered before the write in A. But that write is a release, not an acquire, so it isn't ordered with respect to the subsequent read of X in A; similarly, B's read of inspected is an acquire, not a release, so it isn't ordered with respect to the previous write of X in B. Moreover the read of X in A races with the write in B, and may or may not see its value. Bottom line: a single `volatile` label forces *one* of the four arcs in the "bowtie" figure to follow intuition, but the other three may not.

(Hint: You may find it useful to consult Doug Lea's Java Memory Model "Cookbook for Compiler Writers," at *gee.cs.oswego.edu/dl/jmm/cookbook.html*).

13.10 Implement the nonblocking queue of Example 13.30 on an x86. (Complete pseudocode can be found in the paper by Michael and Scott [MS98].) Do you need fence instructions to ensure consistency? If you have access to appropriate hardware, port your code to a machine with a more relaxed memory model (e.g., ARM or Power). What new fences or atomic references do you need?

Answer:

13.11 Consider the implementation of software transactional memory in Figure 13.19.

(a) How would you implement the read_set, write_map, and lock_map data structures? You will want to minimize the cost not only of insert and lookup operations but also of (1) "zeroing out" the table at the end of a transaction, so it can be used again; and (2) extending the table if it becomes too full.

Answer: In our work at Rochester, we have used closed chaining with a version number in each bucket and another for the table as a whole. Incrementing the whole-table version has the effect of invalidating every entry in the table. If the table gets too full, we allocate a new one twice the size and copy entries from the old to the new. We then keep the larger table for use in future transactions. One could also keep track of average load factor over a series of transactions, and downsize the table if it no longer seems to need to be so large.

(b) The validate routine is called in two different places. Expand these calls inline and customize them to the calling context. What optimizations can you achieve?

Answer: In read, we are guaranteed that the caller has not locked any orecs, so one of the conditional checks is unnecessary. In commit, on the other hand, we no longer need to update valid_time.

(c) Optimize the commit routine to exploit the fact that a final validation is unnecessary if no other transaction has committed since valid_time.

Answer: The following lines of code

```
n : time := 1 + fetch_and_increment(&clock)
validate()
```

should become

```
if (n : time := 1 + fetch_and_increment(&clock)) > 1 + valid_time
    validate()
```

(d) Further optimize commit by observing that the for loop in the finally clause really needs to iterate over orecs, not over addresses (there may be a difference, if more than one address hashes to the same orec). What data, ideally, should lock_map hold?

Answer: Ideally, lock_map should still hold address, orec pairs, but the addresses should be pointers into the orec table, not key values with which to index that table. The following lines of code

```
lock_map[a] := o
...
for ⟨a, o⟩ ∈ lock_map
    orecs[hash(a)] := ...
```

should become

```
lock_map[&orecs[hash(a)]] := o
...
for ⟨a, o⟩ ∈ lock_map
    *a := ...
```

13.12 The code of Example 13.35 could fairly be accused of displaying poor abstraction. If we make *desired_condition* a delegate (a subroutine or object closure), can we pass it as an extra parameter, and move the signal and scheduler_lock management inside sleep_on? (Hint: Consider the code for the P operation in Figure 13.15.)

Answer: Probably not, at least not easily. In the code for P, we need to compare S.N to zero and then decrement it, atomically. In this particular case we might package the decrement inside *desired_condition*, but that would be bad style (evaluating the predicate would have a side effect), and the solution would not generalize to other kinds of synchronization.

13.13 The mechanism used in Figure 13.13 to make scheduler code reentrant employs a single OS-provided lock for all the scheduling data structures of the application. Among other things, this mechanism prevents threads on separate processors from performing P or V operations on unrelated semaphores, even when none of the operations needs to block. Can you devise another synchronization mechanism for scheduler-related operations that admits a higher degree of concurrency but that is still correct?

Answer: There are many subtle variations on scheduler implementation. One simple way to eliminate the window is as follows: In the context block of each thread is a flag that says whether the state of the thread has been saved consistently. Prior to putting its context block into a synchronization data structure, a thread sets the bit to say it is *not* yet saved. In the middle of yield, after the state is saved, we flip the bit the other way. Then when taking a thread off the ready list, we spin until the bit says the state is consistent, prior to restoring it.

13.14 Show how to implement a lock-based concurrent set as a singly linked sorted list. Your implementation should support insert, find, and remove operations, and should permit operations on separate portions of the list to occur concurrently (so a single lock for the entire list will not suffice). (Hint: You will want to use a "walking lock" idiom in which acquire and release operations are interleaved in non-LIFO order.)

Answer:

13.15 (Difficult) Implement a nonblocking version of the set of the previous exercise. (Hint: You will probably discover that insertion is easy but deletion is hard. Consider a *lazy deletion* mechanism in which cleanup [physical removal of a node] may occur well after logical completion of the removal. For further details see the work of Harris [Har01].)

Answer: The key is simply to *mark* a node as deleted and modify all routines to remove marked nodes as they are encountered. Pseudocode can be found in Harris's paper.

13.16 To make spin locks useful on a multiprogrammed multiprocessor, one might want to ensure that no process is ever preempted in the middle of a critical section. That way it would always be safe to spin in user space, because the process holding the lock would be guaranteed to be running on some other processor, rather than preempted and possibly in need of the current processor. Explain why an operating system designer might not want to give user processes the ability to disable preemption arbitrarily. (Hint: Think about fairness and multiple users.) Can you suggest a way to get around the problem? (References to several possible solutions can be found in the paper by Kontothanassis, Wisniewski, and Scott [KWS97].)

Answer:

13.17 Show how to use semaphores to construct a scheduler-based n-thread barrier.

Answer:

13.18 Prove that monitors and semaphores are equally powerful. That is, use each to implement the other. In the monitor-based implementation of semaphores, what is your monitor invariant?

Answer:

13.19 Show how to use binary semaphores to implement general semaphores.

Answer:

13.20 In Example 13.38 (Figure 13.15), suppose we replaced the middle four lines of procedure P with

```
if S.N = 0
    sleep_on(S.Q)
S.N −:= 1
```

and the middle four lines of procedure V with

```
S.N +:= 1
if S.Q is nonempty
    enqueue(ready_list, dequeue(S.Q))
```

What is the problem with this new version? Explain how it connects to the question of hints and absolutes in Section 13.4.1.

Answer:

13.21 Suppose that every monitor has a separate mutual exclusion lock, so that different threads can run in different monitors concurrently, and that we want to release exclusion on both inner and outer monitors when a thread `waits` in a nested call. When the thread awakens it will need to reacquire the outer locks. How can we ensure its ability to do so? (Hint: Think about the order in which to acquire locks, and be prepared to abandon Hoare semantics. For further hints, see Wettstein [Wet78].)

Answer:

13.22 Show how general semaphores can be implemented with conditional critical regions in which all threads wait for the same condition, thereby avoiding the overhead of unproductive wake-ups.

Answer:

13.23 Write code for a bounded buffer using the protected object mechanism of Ada 95.

Answer:

13.24 Repeat the previous exercise in Java using `synchronized` statements or methods. Try to make your solution as simple and conceptually clear as possible. You will probably want to use `notifyAll`.

Answer:

```
class BB {
    final private int SIZE = 10;
    private Object[] buf = new Object[SIZE];

    private int nextEmpty = 0;
    private int nextFull = 0;
    private int fullSlots = 0;

    synchronized public void insert(Object d) throws InterruptedException {
        while (fullSlots == SIZE) {
            wait();
        }
        buf[nextEmpty] = d;
        nextEmpty = (nextEmpty + 1) % SIZE;
        ++fullSlots;
        notifyAll();
    }
```

```
    synchronized public Object remove() throws InterruptedException {
        while (fullSlots == 0) {
            wait();
        }
        Object d = buf[nextFull];
        nextFull = (nextFull + 1) % SIZE;
        --fullSlots;
        notifyAll();
        return d;
    }
}
```

13.25 Give a more efficient solution to the previous exercise that avoids the use of `notifyAll`. (*Warning:* It is tempting to observe that the buffer can never be both full and empty at the same time, and to assume therefore that waiting threads are either all producers or all consumers. This need not be the case, however: if the buffer ever becomes even a temporary performance bottleneck, there may be an arbitrary number of waiting threads, including both producers and consumers.)

Answer:

```
class BB {
    final private int SIZE = 10;
    private Object[] buf = new Object[SIZE];

    private Object producerMutex = new Object();
    // waited upon only by producers
    // protects the following:
        private int nextEmpty = 0;
        private int emptySlots = SIZE;

    private Object consumerMutex = new Object();
    // waited upon only by consumers
    // protects the following:
        private int nextFull = 0;
        private int fullSlots = 0;

    public void insert(Object d) throws InterruptedException {
        synchronized (producerMutex) {
            while (emptySlots == 0) {
                producerMutex.wait();
            }
            --emptySlots;
            buf[nextEmpty] = d;
            nextEmpty = (nextEmpty + 1) % SIZE;
        }
        synchronized (consumerMutex) {
```

```
            ++fullSlots;
            consumerMutex.notify();
        }
    }

    public Object remove() throws InterruptedException {
        Object d;
        synchronized (consumerMutex) {
            while (fullSlots < 0) {
                consumerMutex.wait();
            }
            --fullSlots;
            d = buf[nextFull];
            nextFull = (nextFull + 1) % SIZE;
        }
        synchronized (producerMutex) {
            ++emptySlots;
            producerMutex.notify();
        }
        return d;
    }
}
```

13.26 Repeat the previous exercise using Java `Lock` variables.

Answer:

```
class BB {
    final private int SIZE = 10;
    private Object[] buf = new Object[SIZE];

    private int nextEmpty = 0;
    private int nextFull = 0;
    private int fullSlots = 0;

    Lock l = new ReentrantLock();
    final Condition emptySlot  = l.newCondition();
    final Condition fullSlot = l.newCondition();
```

```
    public void insert(Object d) throws InterruptedException {
        l.lock();
        try {
            while (fullSlots == SIZE) {
                emptySlot.await();
            }
            buf[nextEmpty] = d;
            nextEmpty = (nextEmpty + 1) % SIZE;
            ++fullSlots;
            fullSlot.signal();
        } finally {
            l.unlock();
        }
    }

    public Object remove() throws InterruptedException {
        l.lock();
        try {
            while (fullSlots == 0) {
                fullSlot.await();
            }
            Object d = buf[nextFull];
            nextFull = (nextFull + 1) % SIZE;
            --fullSlots;
            emptySlot.signal();
            return d;
        } finally {
            l.unlock();
        }
    }
}
```

13.27 Explain how *escape analysis*, mentioned briefly in Sidebar 10.3, could be used to reduce the cost of certain `synchronized` statements and methods in Java.

Answer: If the compiler is able to deduce that references to a given object will never escape a given method, then synchronization on that object will never be required, because no other thread will ever be able to see it. While one might at first question whether this observation really matters (after all, who would attempt to use synchronization on a purely local object?), there are many library routines (including the standard `String` library) that make heavy use of `synchronized`. If the compiler knows that an object will always be local, it can arrange to call special versions of those libraries that omit the synchronization.

13.28 The *dining philosophers problem* [Dij72] is a classic exercise in synchronization (Figure 13.20). Five philosophers sit around a circular table. In the center is a large communal plate of spaghetti. Each philosopher repeatedly thinks for a while and then eats for a while, at intervals of his or her own choosing. On the table between each pair of adjacent philosophers is a single fork. To eat, a philosopher requires

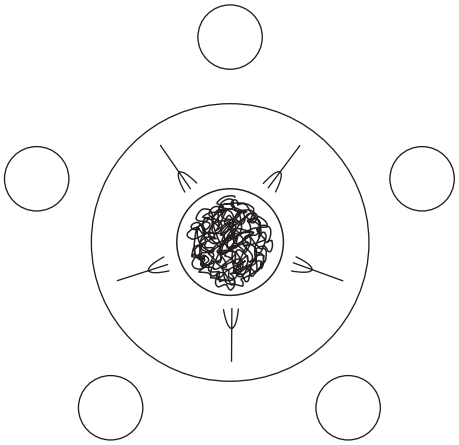


Figure 13.20 **The Dining Philosophers.** Hungry philosophers must contend for the forks to their left and right in order to eat.

both adjacent forks: the one on the left and the one on the right. Because they share a fork, adjacent philosophers cannot eat simultaneously.

Write a solution to the dining philosophers problem in which each philosopher is represented by a process and the forks are represented by shared data. Synchronize access to the forks using semaphores, monitors, or conditional critical regions. Try to maximize concurrency.

Answer:

13.29 In the previous exercise you may have noticed that the dining philosophers are prone to deadlock. One has to worry about the possibility that all five of them will pick up their right-hand forks simultaneously, and then wait forever for their left-hand neighbors to finish eating.

Discuss as many strategies as you can think of to address the deadlock problem. Can you describe a solution in which it is provably impossible for any philosopher to go hungry forever? Can you describe a solution that is fair in a strong sense of the word (i.e., in which no one philosopher gets more chance to eat than some other over the long term)? For a particularly elegant solution, see the paper by Chandy and Misra [CM84].

Answer:

13.30 In some concurrent programming systems, global variables are shared by all threads. In others, each newly created thread has a separate copy of the global variables, commonly initialized to the values of the globals of the creating thread. Under this private globals approach, shared data must be allocated from a special heap. In still other programming systems, the programmer can specify which global variables are to be private and which are to be shared.

Discuss the tradeoffs between private and shared global variables. Which would you prefer to have available, for which sorts of programs? How would you implement each? Are some options harder to implement than others? To what extent do your answers depend on the nature of processes provided by the operating system?

Answer:

13.31 Rewrite Example 13.51 in Java.

Answer:

```
ExecutorService exec = ...
...
Future<String> description = exec.submit(new Callable<String>() {
    public String call() { return getDescription(); }
});
Future<Integer> numberInStock = exec.submit(new Callable<Integer>() {
    public Integer call() { return getInventory(); }
});
...
System.out.println("We have " + numberInStock.get()
    + " copies of " + description.get() + " in stock");
```

Here the standard class `ExecutorService` provides a `submit` method that takes a `Callable` as argument and returns a `Future`.

13.32 AND parallelism in logic languages is analogous to the parallel evaluation of arguments in a functional language (e.g., Multilisp). Does OR parallelism have a similar analog? (Hint: Think about special forms [Section 11.5].) Can you suggest a way to obtain the effect of OR parallelism in Multilisp?

Answer:

13.33 In Section 13.4.5 we claimed that both AND parallelism and OR parallelism were problematic in Prolog, because they failed to adhere to the deterministic search order required by language semantics. Elaborate on this claim. What specifically can go wrong?

Answer:

IN MORE DEPTH

13.34 In Section 13.4.1 we cast monitors as a mechanism for synchronizing access to shared memory, and we described their implementation in terms of semaphores. It is also possible to think of a monitor as a module inhabited by a single thread, which accepts request messages from other threads, performs appropriate operations, and replies. Give the details of a monitor implementation consistent with this conceptual model. Be sure to include condition variables. (Hint: See the discussion of early reply in Section 13.2.3.)

Answer:

13.35 Show how shared memory can be used to implement message passing. Specifically, choose a set of message-passing operations (e.g., no-wait `send` and explicit message receipt) and show how to implement them in your favorite shared-memory notation.

Answer:

13.36 When implementing reliable messages on top of unreliable messages, a sender can wait for an acknowledgment message, and retransmit if it doesn't receive it within a bounded period of time. But how does the receiver know that its acknowledgment has been received? Why doesn't the sender have to acknowledge the acknowledgment (and the receiver acknowledge the acknowledgment of the acknowledgment ...)? (For more information on the design of fast, reliable protocols, you might want to consult a text on computer networks [TW12, PD12].)

Answer: The key observation is that the acknowledgment doesn't carry any new information. In effect, if the acknowledgment doesn't get back to the sender, it's not the receiver's problem: the sender will transmit if *either* the original message *or* the acknowledgment is lost. The only tricky part is to make sure that if the acknowledgment is lost and the sender re-transmits a message that the receiver has really seen, the receiver will be able to determine what has happened, and discard the duplicate message. That's what sequence numbers are for.

13.37 While Go allows both *input* (receive) and *output* (send) guards on its `select` statements, Occam and CSP allow only input guards. The difference has to do with the fact that Go is designed for communication among threads in a single address space, while Occam and CSP were designed for a distributed environment. Why should this make a difference? Suppose you wished to add output guards to Occam. How would the implementation work? (Hint: For ideas, see the article by Bagrodia [Bag86].)

Answer:

13.38 In Section C-13.5.3 we described the semantics of a `terminate` arm on an Ada `select` statement: this arm may be selected if and only if all potential communication partners have terminated, or are likewise stuck in `select` statements with `terminate` arms. Erlang and Occam have no similar facility, though the original CSP proposal does. How would you implement `terminate` arms in Ada? Why do you suppose they were left out of Erlang and Occam? (Hint: For ideas, see the work of Apt and Francez [Fra80, AF84].)

Answer:

14 Scripting Languages

14.7 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

14.1 Does filename "globbing" provide the expressive power of standard regular expressions? Explain.

Answer:

14.2 Write shell scripts to

(a) Replace blanks with underscores in the names of all files in the current directory.

Answer:

(b) Rename every file in the current directory by prepending to its name a textual representation of its modification date.

Answer:

(c) Find all `eps` files in the file hierarchy below the current directory, and create any corresponding `pdf` files that are missing or out of date.

Answer:

(d) Print the names of all files in the file hierarchy below the current directory for which a given predicate evaluates to true. Your (quoted) predicate should be specified on the command line using the syntax of the Unix `test` command, with one or more at signs (`@`) standing in for the name of the candidate file.

Answer:

14.3 In Example 14.16 we used `"$@"` to refer to the parameters passed to `ll`. What would happen if we removed the quote marks? (Hint: Try this for files whose names contain spaces!) Read the `man` page for `bash` and learn the difference between `$@` and `$*`. Create versions of `ll` that use `$*` or `"$*"` instead of `"$@"`. Explain what's going on.

Answer:

14.4 (a) Extend the code in Figure 14.5, 14.6, or 14.7 to try to kill processes more gently. You'll want to read the `man` page for the standard `kill` command. Use a `TERM` signal first. If that doesn't work, ask the user if you should resort to `KILL`.

Answer:

(b) Extend your solution to part (a) so that the script accepts an optional argument specifying the signal to be used. Alternatives to `TERM` and `KILL` include `HUP`, `INT`, `QUIT`, and `ABRT`.

Answer:

14.5 Write a Perl, Python, or Ruby script that creates a simple *concordance*: a sorted list of significant words appearing in an input document, with a sublist for each that indicates the lines on which the word occurs, with up to six words of surrounding context. Exclude from your list all common articles, conjunctions, prepositions, and pronouns.

Answer:

14.6 Write Emacs Lisp scripts to

(a) Insert today's date into the current buffer at the insertion point (current cursor location).

Answer:

(b) Place quote marks (`" "`) around the word surrounding the insertion point.

Answer:

(c) Fix end-of-sentence spaces in the current buffer. Use the following heuristic: if a period, question mark, or exclamation point is followed by a single space (possibly with closing quote marks, parentheses, brackets, or braces in-between), then add an extra space, unless the character preceding the period, question mark, or exclamation point is a capital letter (in which case we assume it is an abbreviation).

Answer:

(d) Run the contents of the current buffer through your favorite spell checker, and create a new buffer containing a list of misspelled words.

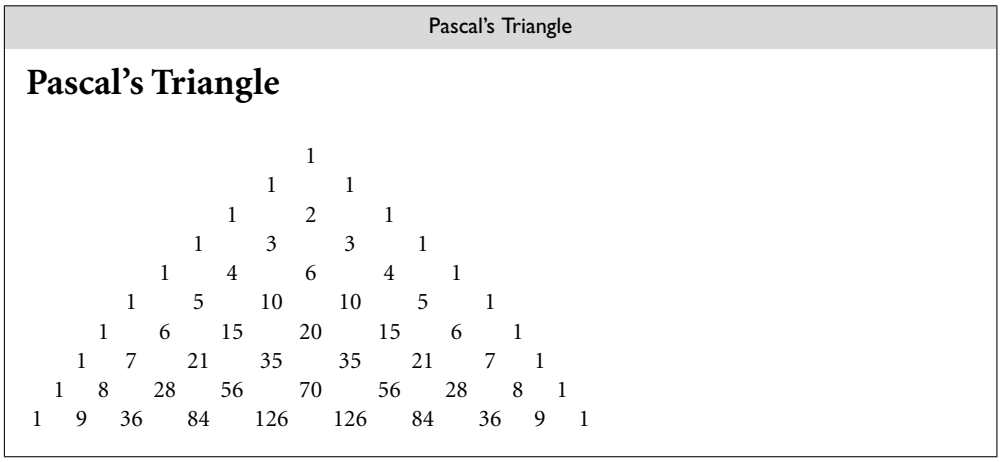


Figure 14.21 Pascal's triangle rendered in a web page (Exercise 14.8).

Answer:

(e) Delete one misspelled word from the buffer created in (d), and place the cursor (insertion point) on top of the first occurrence of that misspelled word in the current buffer.

Answer:

14.7 Explain the circumstances under which it makes sense to realize an interactive task on the Web as a CGI script, an embedded server-side script, or a client-side script. For each of these implementation choices, give three examples of tasks for which it is clearly the preferred approach.

Answer:

14.8 (a) Write a web page with embedded PHP to print the first 10 rows of Pascal's triangle (see Example C-17.10 if you don't know what this is). When rendered, your output should look like Figure 14.21.

Answer:

```
<HTML>
<HEAD>
<TITLE>Pascal's Triangle</TITLE>
</HEAD>
<BODY>
<H1>Pascal's Triangle</H1>
<TABLE>
<?php
    $n = 10;
    for ($r = 0; $r <= $n; $r++) {
        $A[0] = $A[$r] = $t = 1;
```

```
            for ($i = 1; $i <= $r/2; $i++) {
                $A[$i] = $A[$r-$i] = $t = ($t * ($r + 1 - $i)) / $i;
            }
            echo "<TR>";
            for ($i = 0; $i < $n-$r; $i++) echo "<TD>";
            echo "<TD>1";
            for ($i = 1; $i <= $r; $i++) echo "<TD><TD>{$A[$i]}";
            echo "\n";
        }
    ?>
</TABLE>
</BODY>
</HTML>
```

(b) Modify your page to create a self-posting form that accepts the number of desired rows in an input field.

Answer:

(c) Rewrite your page in JavaScript.

Answer:

14.9 Create a fill-in web form that uses a JavaScript implementation of the Luhn formula (Exercise 4.10) to check for typos in credit card numbers. (But don't use real credit card numbers; homework exercises don't tend to be very secure!)

Answer:

14.10 (a) Modify the code of Figure 14.15 (Example 14.35) so that it replaces the form with its output, as the CGI and PHP versions of Figures 14.11 and 14.14 do.

Answer:

(b) Modify the CGI and PHP scripts of Figures 14.11 and 14.14 (Examples 14.30 and 14.34) so they appear to append their output to the bottom of the form, as the JavaScript version of Figure 14.15 does.

Answer:

14.11 Run the following program in Perl:

```
sub foo {
    my $lex = $_[0];
    sub bar {
        print "$lex\n";
    }
    bar();
}

foo(2);  foo(3);
```

You may be surprised by the output. Perl 5 allows named subroutines to nest, but does not create closures for them properly. Rewrite the code above to create a reference to an anonymous local subroutine and verify that it does create closures correctly. Add the line `use diagnostics;` to the beginning of the original version and run it again. Based on the explanation this will give you, speculate as to how nested named subroutines are implemented in Perl 5.

Answer:

14.12 Write a program that will map the web pages stored in the file hierarchy below the current directory. Your output should itself be a web page, containing the names of all directories and `.html` files, printed at levels of indentation corresponding to their level in the file hierarchy. Each `.html` file name should be a live link to its file. Use whatever language(s) seem most appropriate to the task.

Answer:

14.13 In Section 14.4.1 we claimed that nested blocks in Ruby were part of the named scope in which they appear. Verify this claim by running the following Ruby script and explaining its output:

```
def foo(x)
    y = 2
    bar = proc {
        print x, "\n"
        y = 3
    }
    bar.call()
    print y, "\n"
end

foo(3)
```

Now comment out the second line (`y = 2`) and run the script again. Explain what happens. Restate our claim about scoping more carefully and precisely.

Answer:

14.14 Write a Perl script to translate English measurements (in, ft, yd, mi) into metric equivalents (cm, m, km). You may want to learn about the `e` modifier on regular expressions, which allows the right-hand side of an `s///e` expression to contain executable code.

Answer:

14.15 Write a Perl script to find, for each input line, the longest substring that appears at least twice within the line, without overlapping. (*Warning:* This is harder than it sounds. Remember that by default Perl searches for a *left-most longest* match.)

Answer: As usual in Perl, "There's More Than One Way To Do It." Here's one:

```
while (<>) {
    for ($len = int(length($_)/2); $len > 0; $len--) {
        if (/.*(.{$len}).*\1.*/) {
            print "matched $1\n";
            last;       # exit loop
        }
    }
}
```

14.16 Perl provides an alternative `(?:...)` form of parentheses that supports grouping in regular expressions without performing capture. Using this syntax, Example 14.57 could have been written as follows:

```
if (/^([+-]?)((\d+)\.|(\d*)\.(\d+))(?:e([+-]?\d+))?$/) {
    # floating-point number
    print "sign:     ", $1, "\n";
    print "integer:  ", $3, $4, "\n";
    print "fraction: ", $5, "\n";
    print "mantissa: ", $2, "\n";
    print "exponent: ", $6, "\n";           # not $7
}
```

What purpose does this extra notation serve? Why might the code here be preferable to that of Example 14.57?

Answer: Capturing subexpressions takes time. In an inner loop, the cost of unnecessary capture could have a noticeable effect on run time. One can also argue that capturing only what we need makes programs easier to maintain. If we add a new set of noncapturing parentheses to the middle of a complicated expression, we won't have to renumber subsequent uses of `$1`, `$2`, and so on.

14.17 Consider again the `sed` code of Figure 14.1. It is tempting to write the first of the compound statements as follows (note the differences in the three substitution commands):

```
/<[hH][123]>.*<\/[hH][123]>/ {   ;# match whole heading
    h                            ;# save copy of pattern space
    s/^.*\(<[hH][123]>\)/\1/     ;# delete text before opening tag
    s/\(<\/[hH][123]>\).*$/\1/   ;# delete text after closing tag
    p                            ;# print what remains
    g                            ;# retrieve saved pattern space
    s/^.*<\/[hH][123]>//         ;# delete through closing tag
    b top
```

Explain why this doesn't work. (Hint: Remember the difference between *greedy* and *minimal* matches [Example 14.53]. `Sed` lacks the latter.)

Answer: Because regular expressions in `sed` are greedy, the `.*` constructs on lines 3 and 7 will match everything up to the *last* occurrence of an opening tag on a given line of input, *not* the first. The (admittedly contrived) input

```
A <H1>first header</H1> B <H2>second
header</H2> C
<H3>third header</H3> D <H1>fourth</H1> E
```

will yield the output

```
<H2>second
<H2>second
header</H2>
<H1>fourth</H1>
```

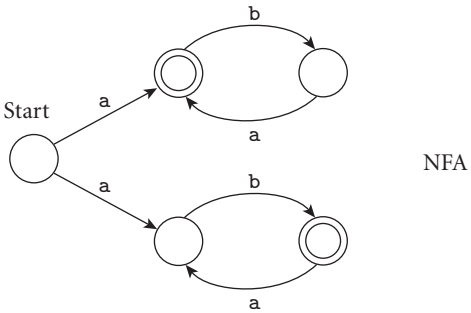
rather than the expected

```
<H1>first header</H1>
<H2>second
header</H2>
<H3>third header</H3>
<H1>fourth</H1>
```

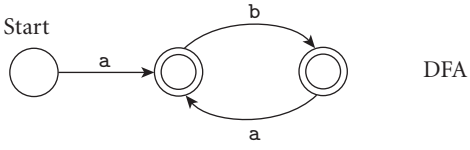
Given the first line of input, the substitution on line 3 of the script, which should presumably delete just `A`, deletes the entire first header instead, leaving nothing for the substitution on line 4 to delete. Similarly, in the third line of input, the third header is skipped in its entirety.

14.18 Consider the following regular expression in Perl: `/^(?:((?:ab)+)|a((?:ba)*))$/`. Describe, in English, the set of strings it will match. Show a natural NFA for this set, together with the minimal DFA. Describe the substrings that should be captured in each matching string. Based on this example, discuss the practicality of using DFAs to match strings in Perl.

Answer: This RE matches all strings of alternating `a`s and `b`s, beginning with an `a`.



NFA



DFA

If the string ends with a `b`, `$1` should capture the entire string. If the string ends with a `a`, `$2` should capture all but the initial `a`. It is easy to build these captured strings incrementally during a backtracking emulation of the NFA. There is no obvious way to build them during emulation of the DFA. This is one of the principal reasons why Perl uses NFAs.

IN MORE DEPTH

14.19 Modify the XSLT of Figure C-14.24 to do one or more of the following:

(a) Alter the titles of conference papers so that only first words, words that follow a dash or colon (and thus begin a subtitle), and proper nouns are capitalized. You will need to adopt a convention by which the creator of the document can identify proper nouns.

Answer:

(b) Sort entries by the last name of the first author or editor. You will need to adopt a convention by which the creator of the document can identify compound last names ("von Neumann," for example, should be alphabetized under 'v').

Answer:

(c) Allow bibliographic entries to contain an `abstract` element, which when formatted appears as an indented block of text in a smaller font.

Answer:

(d) In addition to the `book`, `article`, and `inproceedings` elements, add support for other kinds of entries, such as manuals, technical reports, theses, newspaper articles, web sites, and so on. You may want to draw inspiration from the categories supported by BibTeX [Lam94, App. B].

Answer:

(e) Format entries according to some standard style convention (e.g., that of the Chicago Manual of Style [*www.chicagomanualofstyle.org/16/ch14/ch14_toc.html*] or the ACM Transactions [*www.acm.org/publications/article-templates/acm-latex-style-guide*]).

Answer:

14.20 Suppose bibliographic entries in Figure C-14.23 contain a mandatory `key` element, and that other documents can contain matching `cite` elements. Create an XSLT script that imitates the work of BibTeX. Your script should

(a) read an XML document, find all the `cite` elements, collect the keys they contain, and replace them with `bibref` elements that contain small integers instead.

(b) read a separate XML bibliography document, extract the entries with matching keys, and write them, in sorted order, to a new (and probably smaller) bibliography.

The small numbers in the `bibref` elements of the new document from (a) should match the corresponding numbered entries in the new bibliography from (b).

Answer:

14.21 Write a program that will read an XHTML file and print an outline of its contents, by extracting all `<title>`, `<h1>`, `<h2>`, and `<h3>` elements, and printing them at varying levels of indentation. Write

(a) in C or Java

(b) in `sed` or `awk`

(c) in Perl, Python, Tcl, or Ruby

(d) in XSLT

Compare and contrast your solutions.

Answer:

15 Building a Runnable Program

15.10 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

15.1 If you were writing a two-pass compiler, why might you choose a high-level IF as the link between the front end and the back end? Why might you choose a medium-level IF?

Answer:

15.2 Consider a language like Ada or Modula-2, in which a module *M* can be divided into a specification (header) file and an implementation (body) file for the purpose of separate compilation (Section 10.2.1). Should *M*'s specification itself be separately compiled, or should the compiler simply read it in the process of compiling *M*'s body and the bodies of other modules that use abstractions defined in *M*? If the specification is compiled, what should the output consist of?

Answer: There is a simple tradeoff here between execution speed and complexity of implementation. The simplest way to handle specification modules is to expand them textually at the beginning of all appropriate implementation modules, much as `.h` files are `#included` in `.c` files in C. The problem with this approach is that the same source ends up being compiled multiple times, which is inefficient. If we choose to compile specification modules instead we introduce complexity into the implementation, but save compilation time. The obvious format for a compiled specification module is something that can be incorporated into the compiler's internal symbol table with minimal processing.

15.3 Many research compilers (e.g., for SR [AO93], Cedar [SZBH86], Lynx [Sco91], and Modula-3 [Har92]) have used C as their IF. C is well documented and mostly machine independent, and C compilers are much more widely available than alternative back ends. What are the disadvantages of generating C, and how might they be overcome?

Answer: C lacks many features of the languages that are compiled into it. Examples include nested procedures, true exceptions, and concurrency. There are two general approaches to implementing these features.

(a) Use as much of the expressiveness of C as possible—use the features of C that most closely approximate what the target program needs. This approach often requires non-portable knowledge about how the C compiler implements its features—how it lays out its stack frames for example, or how it returns values from functions. It also tends to produce very baroque code, which doesn't optimize well.

(b) Use only the lowest-level parts of C, as a sort of portable assembler. This approach has the potential to produce better code, by running an optimizer *before* calling the C compiler. It is complicated by the fact that standard C lacks some of the facilities provided by assembly languages, such as the ability to take the address of a label.

15.4 List as many ways as you can think of in which the back end of a just-in-time compiler might differ from that of a more conventional compiler. What design goals dictate the differences?

Answer:

15.5 Suppose that k (the number of temporary registers) in Figure 15.6 is 4 (this is an artificially small number for modern machines). Give an example of an expression that will lead to register spilling under our naive register allocation algorithm.

Answer:

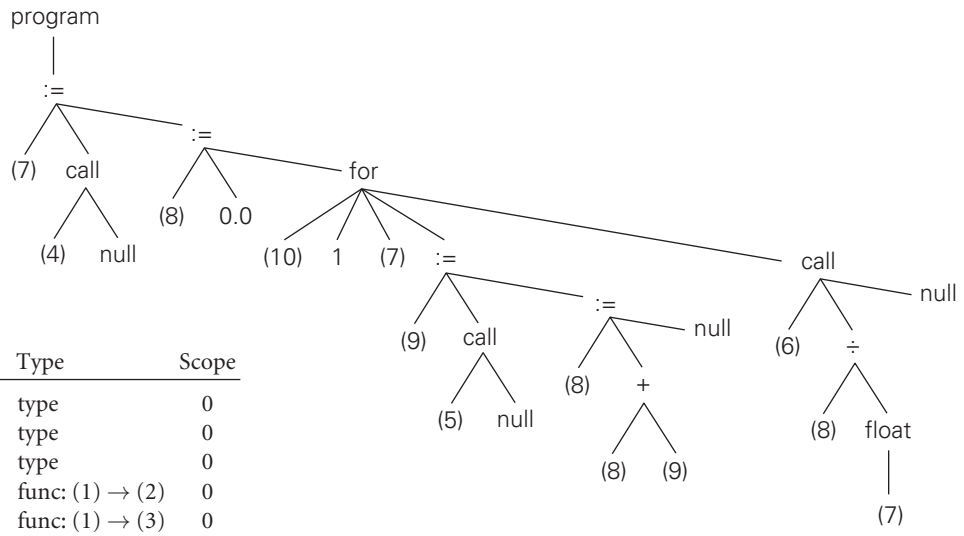
15.6 Modify the attribute grammar of Figure 15.6 in such a way that it will generate the control flow graph of Figure 15.3 instead of the linear assembly code of Figure 15.7.

Answer:

15.7 Add productions and attribute rules to the grammar of Figure 15.6 to handle Ada-style `for` loops (described in Section 6.5.1). Using your modified grammar, hand-translate the syntax tree of Figure 15.10 into pseudo-assembly notation. Keep the index variable and the upper loop bound in registers.

Answer:

15.8 One problem (of many) with the code we generated in Section 15.3 is that it computes at run time the value of expressions that could have been computed at compile time. Modify the grammar of Figure 15.6 to perform a simple form of *constant folding*: whenever both operands of an operator are compile-time constants, we should compute the value at compile time and then generate code that uses the value directly. Be sure to consider how to handle overflow.



Index	Symbol	Type	Scope
1	`void`	type	0
2	`int`	type	0
3	`real`	type	0
4	`getint`	func: (1) → (2)	0
5	`getreal`	func: (1) → (3)	0
6	`putreal`	func: (3) → (1)	0
7	`n`	2	1
8	`sum`	3	1
9	`x`	3	1
10	`i`	2	2

Figure 15.10 Syntax tree and symbol table for a program that computes the average of `N` real numbers. The children of the for node are the index variable, the lower bound, the upper bound, and the body.

Answer:

15.9 Modify the grammar of Figure 15.6 to generate jump code for Boolean expressions, as described in Section 6.4.1. You should assume short-circuit evaluation (Section 6.1.5).

Answer:

15.10 Our GCD program did not employ subroutines. Extend the grammar of Figure 15.6 to handle procedures without parameters (feel free to adopt any reasonable conventions on the structure of the syntax tree). Be sure to generate appropriate prologue and epilogue code for each subroutine, and to save and restore any needed temporary registers.

Answer:

15.11 The grammar of Figure 15.6 assumes that all variables are global. In the presence of subroutines, we should need to generate different code (with `fp`-relative displacement mode addressing) to access local variables and parameters. In a language with nested scopes we should need to dereference the static chain (or index into the display) to access objects that are neither local nor global. Suppose that we are compiling a language with nested subroutines, and are using a static chain. Modify

the grammar of Figure 15.6 to generate code to access objects correctly, regardless of scope. You may find it useful to define a to_register subroutine that generates the code to load a given object. Be sure to consider both l-values and r-values, and parameters passed by both value and result.

Answer:

IN MORE DEPTH

15.12 Compare and contrast GIMPLE with the notation we have been using for syntax tree attribute grammars (Section 4.6).

Answer:

15.13 PC-relative branches on many processors are limited in range—they can only target locations within 2^k bytes of the current PC, for some k less than the wordsize of the machine. Explain how to generate position-independent code that needs to branch farther than this.

Answer: There are at least three alternatives. As noted in Example 15.16, all control transfers on an ARM processor are PC-relative, so limited range may be an issue even in non-PIC code. The solution on ARM is to issue a multi-instruction sequence adds one or more constants to the PC, storing the result in a temporary register (usually `r12`), and then branches indirectly through the register.

A second option, which avoids the need for a spare register but is rather more cumbersome to implement, is to create a series of "stepping stones" in code: the branch jumps not to its actual destination, but to the first of the stepping stones. This in turn branches to the second, and so on. Ideally, the stepping stones will be placed immediately after unconditional branches in the code, where they have no negative impact. If appropriate locations cannot be found, they can be placed in arbitrary places, proceeded with an unconditional branch that hops over them.

A third alternative is to use an indirect branch through a location in the linkage table. In effect, this treats each long-distance branch as if it were dynamically linked.

15.14 We have noted that Linux creates a single data segment containing all the static data of libraries that might be called (directly or indirectly) by a given program. The space required for this segment is usually not a problem: most libraries have little static data—often none at all. Suppose this were not the case. If we wanted to perform dynamic linking for modules with large amounts of per-module static data, how could we extend Linux's dynamic linking mechanisms to perform fully dynamic (lazy) linking not only of code, but also of data?

Answer: Previous editions of this textbook described the dynamic linking of MIPS/IRIX systems, which employed one possible solution. The first step is to create a separate data segment for each dynamic library. At load time, the program begins with the main code segment and linkage table, and with all data segments for which addresses need to appear in that linkage table. In the example of Figure C-15.13, we would load the data segments of both `main` and `foo`, because the addresses of both `X` (which belongs to `main`) and `Y` (which belongs to `foo`) need to appear in the main linkage table. We would not, however, load the code segment or linkage table of `foo`,

despite the fact that the address of `foo` needs to appear in the linkage table. As in standard Linux, we would initialize that linkage table entry to refer to a stub routine. When the initial call to `foo` transferred through the stub into the dynamic linker, we would load (as before) the code/PLT and GOT of `foo`. We would also load the data segments of any libraries (e.g., `bar`) for which addresses needed to appear in `foo`'s GOT.

As execution proceeds, further references to not-yet-loaded symbols will extend the "frontier" of the program. Because invocations of the linker occur on subroutine calls and not on data references, the current frontier always includes a set of code segments and the data segments to which those code segments refer. Each linking operation brings in one new code segment, together with all of the additional data segments to which that code refers.

15.15 In Example C-9.61 we described how the GNU Ada Translator (`gnat`) for the x86 uses dynamically generated code to represent a subroutine closure. Explain how a similar technique could be used to simplify the mechanism of Figure C-15.13, if we were willing to modify code segments at run time.

Answer: Consider the call to `bar`. Instead of updating the GOT to replace the address of the `push` instruction in `bar_stub` with the address of `bar` itself, the dynamic linker could patch the code of `bar_stub` to call `bar` directly. This would require, of course, that each process have its own copy of `foo`'s PLT. It would also introduce problems like those discussed in Sidebar C-9.10.

When `main` calls `foo`, we could also consider replacing the call to `foo_stub` with a call to `foo` itself (assuming the two instructions are the same length, or can be padded to be so). This option is not feasible for `foo`'s call to `bar`, however, since `foo`'s code is shared among processes.

16 Run-Time Program Management

16.6 Solutions Manual

This manual contains suggested solutions to many of the PLP exercises. It is provided *only* to instructors who have adopted the text in their course.[1]

16.1 Write the formula of Example 15.4 as an expression tree (a syntax tree in which each operator is represented by an internal node whose children are its operands). Convert your tree to an *expression DAG* by merging identical nodes. Comment on the redundancy in the tree and how it relates to Figure 15.4.

Answer:

16.2 We assumed in Example 15.4 and Figure 15.4 that `a`, `b`, `c`, and `s` were all among the first few local variables of the current method, and could be pushed onto or popped from the operand stack with a single one-byte instruction. Suppose that this is not the case: that is, that the push and pop instructions require three bytes each. How many bytes will now be required for the code on the left side of Figure 15.4?

Most stack-based languages, Java bytecode and CIL among them, provide a `swap` instruction that reverses the order of the top two values on the stack, and a `duplicate` instruction that pushes a second copy of the value currently at top of stack. Show how to use `swap` and `duplicate` to eliminate the `pop` and the `push`es of `s` in the left side of Figure 15.4. Feel free to exploit the associativity of multiplication. How many instructions is your new sequence? How many bytes?

Answer:

16.3 The speculative optimization of Example 16.5 could in principle be statically performed. Explain why a dynamic compiler might be able to do it more effectively.

Answer: A dynamic compiler has at least two advantages. First, it has a better chance (based on profiling) of identifying the functions for which speculative optimization is worthwhile. Second, it can generate only those alternative versions of the code that seem likely to be profitable. Suppose, for example, that we have 20 different classes derived from `C`, three of which (according to profiling data) are frequently passed to `f`. The dynamic compiler can produce the equivalent of the following.

```
static void f(C o) {
    switch (o.getClass()) {
        case D.class: { /* code with inlined calls */ }
        case E.class: { /* code with inlined calls */ }
        case F.class: { /* code with inlined calls */ }
        default:      { /* code without inlined calls */ }
    }
}
```

A traditional compiler that wanted to enable arbitrary run-time inlining would need to generate 20 different versions of the code. [NB: The code shown here is not valid Java: the `switch` statement is applicable only to `integer` values. The compiler is not generating Java, however: it's generating machine code, and can easily conjure up something equivalent.]

16.4 Perhaps the most common form of run-time instrumentation counts the the number of times that each basic block is executed. Since basic blocks are short, adding a load-increment-store instruction sequence to each block can have a significant impact on run time.

We can improve performance by noting that certain blocks imply the execution of other blocks. In an if... then ... else construct, for example, execution of either the then part or the else part implies execution of the conditional test. If we're smart, we won't actually have to instrument the test.

Describe a general technique to minimize the number of blocks that must be instrumented to allow a post-processor to obtain an accurate count for each block. (This is a difficult problem. For hints, see the paper by Larus and Ball [BL92].)

Answer:

16.5 Visit *software.intel.com/en-us/articles/pintool-downloads* and download a copy of Pin. Use it to create a tool to profile loops. When given a (machine code) program and its input, the output of the tool should list the number of times that each loop was encountered when running the program. It should also give a histogram, for each loop, of the number of iterations executed.

Answer:

16.6 Outline mechanisms that might be used by a binary rewriter, without access to source code, to catch uses of uninitialized variables, "double deletes," and uses of deallocated memory (e.g., dangling pointers). Under what circumstances might you be able to catch memory leaks and out-of-bounds array accesses?

Answer:

16.7 Extend the code of Figure 16.4 to print information about

(a) fields
(b) constructors
(c) nested classes
(d) implemented interfaces
(e) ancestor classes, and their methods, fields, and constructors
(f) exceptions thrown by methods
(g) generic type parameters

Answer:

16.8 Repeat the previous exercise in C#. Add information about parameter names and generic instances.

Answer:

16.9 Write an interactive tool that accepts keyboard commands to load specified class files, create instances of their classes, invoke their methods, and read and write their fields. Feel free to limit keyboard input to values of built-in types, and to work only in the global scope. Based on your experience, comment on the feasibility of writing a command-line interpreter for Java, similar to those commonly used for Lisp, Prolog, or the various scripting languages.

Answer:

16.10 In Java, if the concrete type of `p` is `Foo`, `p.getClass()` and `Foo.class` will return the same thing. Explain why a similar equivalence could not be guaranteed to hold in Ruby, Python, or JavaScript. For hints, see Section 14.4.4.

Answer: Ruby and Python allow members to be added or changed on an object-by-object basis; an object cannot be guaranteed to have (only) the default members of its class. JavaScript goes even farther: as an *object-based* language, it has no notion of class.

16.11 Design a "test harness" system based on Java annotations. The user should be able to attach to a method an annotation that specifies parameters to be passed to a test run of the method, and values expected to be returned. For simplicity, you may assume that parameters and return values will all be strings or instances of built-in types. Using the annotation processing facility of Java 6, you should automatically generate a new method, `test()` in any class that has methods with `@Test` annotations. This method should call the annotated methods with the specified parameters, test the return values, and report any discrepancies. It should also call the `test` methods of any nested classes. Be sure to include a mechanism to invoke the `test` method of every top-level class. For an extra challenge, devise a way to specify multiple tests of a single method, and a way to test exceptions thrown, in addition to values returned.

Answer:

16.12 C++ provides a `typeid` operator that can be used to query the concrete type of a pointer or reference variable:

```
if (typeid(*p) == typeid(my_derived_type)) ...
```

Values returned by `typeid` can be compared for equality but not assigned. They also support a `name()` method that returns an (implementation-dependent) character string name for the type. Give an example of a program fragment in which these mechanisms might reasonably be used.

Unlike more extensive reflection mechanisms, `typeid` can be applied only to (instances of) classes with at least one virtual method. Give a plausible explanation for this restriction.

Answer:

16.13 Suppose we wish, as described at the end of Example 16.36, to accurately attribute sampled time to the various contexts in which a subroutine is called. Perhaps the most straightforward approach would be to log not only the current PC but also the stack *backtrace*—the contents of the dynamic chain—on every timer interrupt. Unfortunately, this can dramatically increase profiling overhead. Suggest an equivalent but cheaper implementation.

Answer: At compile or link time, we can build an explicit representation of the program's *call graph*, as described in Exercise 3.10. We can then assign a unique identifier to every path from the root (`main`) to a given node in the graph. (This is a little tricky in programs with recursion; we need to treat as identical any paths that differ only in the number of iterations of a cycle.) Now when procedure `A` calls procedure `B` in the instrumented program, it passes an extra hidden argument that names the path it followed in the call graph. (To compute this argument, it performs a table lookup using the path that was passed it by its caller.) The handler for timer signals now needs to log only the PC and the path identifier.

IN MORE DEPTH

16.14 Using your local implementations of Java and C#, compile the code of Figures 16.2 and C-16.7 all the way to machine language. Disassemble and compare the results. Can all the differences be attributed to variations in the quality of the compilers, or are any reflective of more fundamental differences between the source languages or virtual machines?

Answer:

16.15 Rewrite the list insertion method of Example C-16.40 in F# instead of C#. Compile to CIL and compare to the right side of Figure C-16.7. Discuss any differences you find.

Answer:

16.16 Building on the previous exercise, rewrite your list insertion routine (both C# and F# versions) to be generic in the type of the list elements. Compare the generic and nongeneric versions of the resulting CIL and discuss the differences.

16.17 Extend your F# code from Exercise C-16.16 to include list removal and search routines. After finding and reading appropriate documentation, package these routines in a library that can be called in a natural way not only from F# but also from C#.