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.

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 Translate the following expression into postfix and prefix notation:

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

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`.

6.4 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.

6.5 Example 6.31 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 (Sparc, MIPS, Alpha, PowerPC, ...) 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;
```

7 Data Types

7.13 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.

7.1 Most modern Algol-family languages 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 and SR) 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:

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 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 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:

(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;
```

7.16 In Section 7.4 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:

7.17 Explain how to extend Figure 7.7 to accommodate subroutine arguments that are passed by value, but whose shape is not known until the subroutine is called at run time.

Answer:

7.18 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:

7.19 Consider the following C declaration, compiled on a 32-bit Pentium 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: $1000 + (3 \times 10 \times 8) + (7 \times 8) = 1296$.

7.20 Consider the following Pascal variable declarations:

```
var A : array [1..10, 10..100] of real;
    i : integer;
    x : real;
```

Assume that a real number occupies eight bytes and that `A`, `i`, and `x` are global variables. In something resembling assembly language for a RISC machine, show the code that a reasonable compiler would 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
```

7.21 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: 3in

(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.

7.22 In Examples 7.69 and 7.70, show the code that would be required to access `A[i, j, k]` if subscript bounds checking were required.

Answer:

7.23 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 ends at the same built-in type when used to dive into the structure of B (and encounters the same field names along the way). 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 2.13 [page 53]; details can be found in any automata theory textbook [e.g., [HMU01]].)

Answer:

7.24 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
```

7.25 In Ada 83, as in Pascal, 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?