Fortran, for example, we can limit the use of `gotos` to patterns that mimic the control flow of more modern languages. In languages without short-circuit evaluation, we can write nested selection statements. In languages without iterators, we can write sets of subroutines that provide equivalent functionality.

6.9 Exercises

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?

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?

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?

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.

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.)

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

6.7 In Section 6.1.2 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?

troduced at least in part for the sake of efficient implementation. These include `packed` types, multilength numeric types, `with` statements, decimal arithmetic, and C-style pointer arithmetic.

At the same time, one can identify a growing willingness on the part of language designers and users to tolerate complexity and cost in language implementation in order to improve semantics. Examples here include the type-safe variant records of Ada; the standard-length numeric types of Java and C#; the variable-length strings and string operators of Icon, Java, and C#; the late binding of array bounds in Ada; and the wealth of whole-array and slice-based array operations in Fortran 90. One might also include the polymorphic type inference of ML. Certainly one should include the trend toward automatic garbage collection. Once considered too expensive for production-quality imperative languages, garbage collection is now standard not only in such experimental languages as Clu and Cedar, but in Ada, Modula-3, Java, and C# as well. Many of these features, including variable-length strings, slices, and garbage collection, have been embraced by scripting languages.

7.12 Exercises

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?

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
```

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.

7.14 We noted in Section 7.3.4 that Pascal and Ada require 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?

7.15 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'`.

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.

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.

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.

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]`?

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.

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?

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.

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]].)

7.24 Explain the meaning of the following C declarations.

```
double *a[n];
double (*b)[n];
```

```
double (*c[n])();
double (*d())[n];
```

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?

7.26 As noted in Section 7.7.2, 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?

7.27 In the discussion of pointers in Section 7.7, 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, how would your task be complicated by the existence of such "internal pointers"?

7.28 (a) A tracing garbage collector in a typesafe language can find and reclaim all *unreachable* objects. It will not necessarily reclaim all *useless* objects—those that will never be used again. Explain.

(b) With future technology, might it be possible to design a garbage collector that will reclaim all useless objects? Again, explain.

7.29 (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.

(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?

7.30 In Example 7.96 we noted that reference counts can be used to reclaim tombstones, failing only when the programmer neglects to manually delete