Chapter 7

Code Shape

CHAPTER OVERVIEW

To translate an application program, the compiler must map each source-language statement into a sequence of one or more operations in the target machine's instruction set. The compiler must chose among many alternative ways to implement each construct. Those choices have a strong and direct impact on the quality of the code that the compiler eventually produces.

This chapter explores some of the implementation strategies that the compiler can employ for a variety of common programming-language constructs.

Keywords: Code Generation, Control Structures, Expression Evaluation

7.1 INTRODUCTION

When the compiler translates application code into executable form, it faces myriad choices about specific details, such as the organization of the computation and the location of data. Such decisions often affect the performance of the resulting code. The compiler's decisions are guided by information that it derives over the course of translation. When information is discovered in one pass and used in another, the compiler must record that information for its own later use.

Often, compilers encode facts in the IR form of the program—facts that are hard to re-derive unless they are encoded. For example, the compiler might generate the IR so that every scalar variable that can safely reside in a register is stored in a virtual register. In this scheme, the register allocator's job is to decide which virtual registers it should demote to memory. The alternative, generating the IR with scalar variables stored in memory and having the allocator promote them into registers, requires much more complex analysis.

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00007-4
Copyright © 2012, Elsevier Inc. All rights reserved.

Encoding knowledge into the IR name space in this way both simplifies the later passes and improves the compiler's effectiveness and efficiency.

Conceptual Roadmap

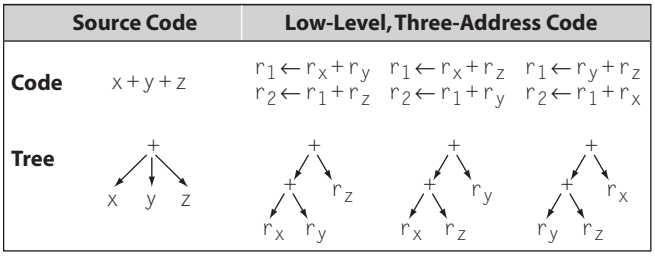
The translation of source code constructs into target-machine operations is one of the fundamental acts of compilation. The compiler must produce target code for each source-language construct. Many of the same issues arise when generating IR in the compiler's front end and generating assembly code for a real processor in its back end. The target processor may, due to finite resources and idiosyncratic features, present a more difficult problem, but the principles are the same.

This chapter focuses on ways to implement various source-language constructs. In many cases, specific details of the implementation affect the compiler's ability to analyze and to improve the code in later passes. The concept of "code shape" encapsulates all of the decisions, large and small, that the compiler writer makes about how to represent the computation in both IR and assembly code. Careful attention to code shape can both simplify the task of analyzing and improving the code, and improve the quality of the final code that the compiler produces.

Overview

In general, the compiler writer should focus on shaping the code so that the various passes in the compiler can combine to produce outstanding code. In practice, a compiler can implement most source-language constructs many ways on a given processor. These variations use different operations and different approaches. Some of these implementations are faster than others; some use less memory; some use fewer registers; some might consume less energy during execution. We consider these differences to be matters of *code shape.*

Code shape has a strong impact both on the behavior of the compiled code and on the ability of the optimizer and back end to improve it. Consider, for example, the way that a C compiler might implement a `switch` statement that switched on a single-byte character value. The compiler might use a cascaded series of `if–then–else` statements to implement the `switch` statement. Depending on the layout of the tests, this could produce different results. If the first test is for zero, the second for one, and so on, then this approach devolves to linear search over a field of 256 keys. If characters are uniformly distributed, the character searches will require an average of 128 tests and branches per character—an expensive way to implement a case statement. If, instead, the tests perform a binary search, the average case would involve eight tests and branches, a more palatable number. To trade



■ **FIGURE 7.1** Alternate Code Shapes for x + y + z.

data space for speed, the compiler can construct a table of 256 labels and interpret the character by loading the corresponding table entry and jumping to it—with a constant overhead per character.

All of these are legal implementations of the `switch` statement. Deciding which one makes sense for a particular `switch` statement depends on many factors. In particular, the number of cases and their relative execution frequencies are important, as is detailed knowledge of the cost structure for branching on the processor. Even when the compiler cannot determine the information that it needs to make the best choice, it must make a choice. The differences among the possible implementations, and the compiler's choice, are matters of code shape.

As another example, consider the simple expression `x + y + z`, where `x`, `y`, and `z` are integers. Figure 7.1 shows several ways of implementing this expression. In source-code form, we may think of the operation as a ternary add, shown on the left. However, mapping this idealized operation into a sequence of binary additions exposes the impact of evaluation order. The three versions on the right show three possible evaluation orders, both as three-address code and as abstract syntax trees. (We assume that each variable is in an appropriately named register and that the source language does not specify the evaluation order for such an expression.) Because integer addition is both commutative and associative, all the orders are equivalent; the compiler must choose one to implement.

Left associativity would produce the first binary tree. This tree seems "natural" in that left associativity corresponds to our left-to-right reading style. Consider what happens if we replace `y` with the literal constant `2` and `z` with `3`. Of course, `x + 2 + 3` is equivalent to `x + 5`. The compiler should detect the computation of `2 + 3`, evaluate it, and fold the result directly into the code. In the left-associative form, however, `2 + 3` never occurs. The order `x + z + y` hides it, as well. The right-associative version exposes the opportunity for

improvement. For each prospective tree, however, there is an assignment of variables and constants to x, y, and z that does not expose the constant expression for optimization.

As with the switch statement, the compiler cannot choose the best shape for this expression without understanding the context in which it appears. If, for example, the expression x + y has been computed recently and neither the values of x nor y have changed, then using the leftmost shape would let the compiler replace the first operation, $r_1 \leftarrow r_x + r_y$, with a reference to the previously computed value. Often, the best evaluation order depends on context from the surrounding code.

This chapter explores the code-shape issues that arise in implementing many common source-language constructs. It focuses on the code that should be generated for specific constructs, while largely ignoring the algorithms required to pick specific assembly-language instructions. The issues of instruction selection, register allocation, and instruction scheduling are treated separately, in later chapters.

7.2 ASSIGNING STORAGE LOCATIONS

As part of translation, the compiler must assign a storage location to each value produced by the code. The compiler must understand the value's type, its size, its visibility, and its lifetime. The compiler must take into account the runtime layout of memory, any source-language constraints on the layout of data areas and data structures, and any target-processor constraints on placement or use of data. The compiler addresses these issues by defining and following a set of conventions.

A typical procedure computes many values. Some of them, such as variables in an Algol-like language, have explicit names in the source code. Other values have implicit names, such as the value i - 3 in the expression A[i - 3, j + 2].

- The lifetime of a named value is defined by source-language rules and actual use in the code. For example, a static variable's value must be preserved across multiple invocations of its defining procedure, while a local variable of the same procedure is only needed from its first definition to its last use in each invocation.
- In contrast, the compiler has more freedom in how it treats unnamed values, such as i - 3. It must handle them in ways that are consistent with the meaning of the program, but it has great leeway in determining where these values reside and how long to retain them.

Compilation options may also affect placement; for example, code compiled to work with a debugger should preserve all values that the debugger can name—typically named variables.

The compiler must also decide, for each value, whether to keep it in a register or to keep it in memory. In general, compilers adopt a "memory model"—a set of rules to guide it in choosing locations for values. Two common policies are a memory-to-memory model and a register-to-register model. The choice between them has a major impact on the code that the compiler produces.

With a memory-to-memory model, the compiler assumes that all values reside in memory. Values are loaded into registers as needed, but the code stores them back to memory after each definition. In a memory-to-memory model, the IR typically uses *physical register* names. The compiler ensures that demand for registers does not exceed supply at each statement.

Physical register
a named register in the target ISA

In a register-to-register model, the compiler assumes that it has enough registers to express the computation. It invents a distinct name, a *virtual register*, for each value that can legally reside in a register. The compiled code will store a virtual register's value to memory only when absolutely necessary, such as when it is passed as a parameter or a return value, or when the register allocator spills it.

Virtual register
a symbolic name used in the IR in place of a physical register name

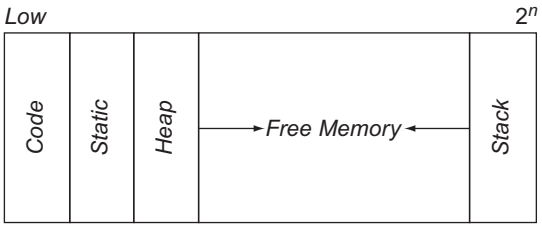
Choice of memory model also affects the compiler's structure. For example, in a memory-to-memory model, the register allocator is an optimization that improves the code. In a register-to-register memory model, the register allocator is a mandatory phase that reduces demand for registers and maps the virtual register names onto physical register names.

7.2.1 Placing Runtime Data Structures

To perform storage assignment, the compiler must understand the system-wide conventions on memory allocation and use. The compiler, the operating system, and the processor cooperate to ensure that multiple programs can execute safely on an interleaved (time-sliced) basis. Thus, many of the decisions about how to lay out, manipulate, and manage a program's address space lie outside the purview of the compiler writer. However, the decisions have a strong impact on the code that the compiler generates. Thus, the compiler writer must have a broad understanding of these issues.

Figure 7.2 shows a typical layout for the address space used by a single compiled program. The layout places fixed size regions of code and data at the low end of the address space. Code sits at the bottom of the address space; the adjacent region, labelled *Static*, holds both static and global data areas, along with any fixed size data created by the compiler. The region above

The compiler may create additional static data areas to hold constant values, jump tables, and debugging information.



■ **FIGURE 7.2** Logical Address-Space Layout.

these static data areas is devoted to data areas that expand and contract. If the compiler can stack-allocate ARs, it will need a runtime stack. In most languages, it will also need a heap for dynamically allocated data structures. To allow for efficient space utilization, the heap and the stack should be placed at opposite ends of the open space and grow towards each other. In the drawing, the heap grows toward higher addresses, while the stack grows toward lower addresses. The opposite arrangement works equally well.

From the compiler's perspective, this logical address space is the whole picture. However, modern computer systems typically execute many programs in an interleaved fashion. The operating system maps multiple logical address spaces into the single physical address space supported by the processor. Figure 7.3 shows this larger picture. Each program is isolated in its own logical address space; each can behave as if it has its own machine.

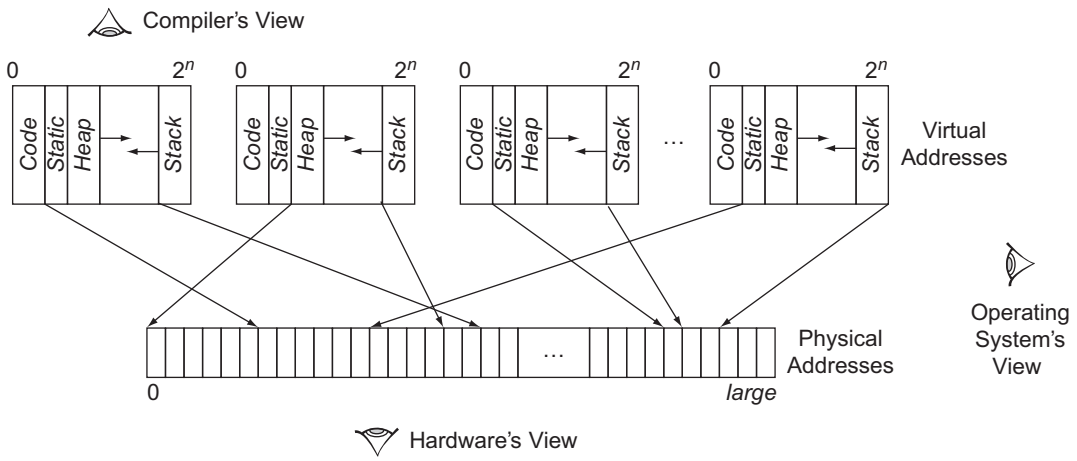
Page
the unit of allocation in a virtual address space

The operating system maps virtual pages into physical page frames.

A single logical address space can occupy disjoint pages in the physical address space; thus, the addresses 100,000 and 200,000 in the program's logical address space need not be 100,000 bytes apart in physical memory. In fact, the physical address associated with the logical address 100,000 may be larger than the physical address associated with the logical address 200,000. The mapping from logical addresses to physical addresses is maintained cooperatively by the hardware and the operating system. It is, in almost all respects, beyond the compiler's purview.

7.2.2 Layout for Data Areas

For convenience, the compiler groups together the storage for values with the same lifetimes and visibility; it creates distinct data areas for them. The placement of these data areas depends on language rules about lifetimes and visibility of values. For example, the compiler can place procedure-local automatic storage inside the procedure's activation record, precisely because the lifetimes of such variables matches the AR's lifetime. In contrast, it must place procedure-local static storage where it will exist across invocations—in the "static" region of memory. Figure 7.4 shows a typical



■ **FIGURE 7.3** Different Views of the Address Space.

```
if x is declared locally in procedure p, and
   its value is not preserved across distinct invocations of p
       then assign it to procedure-local storage
   if its value is preserved across invocations of p
        then assign it to procedure-local static storage
if x is declared as globally visible
        then assign it to global storage
if x is allocated under program control
        then assign it to the runtime heap
```

■ **FIGURE 7.4** Assigning Names to Data Areas.

set of rules for assigning a variable to a specific data area. Object-oriented languages follow different rules, but the problems are no more complex.

Placing local automatic variables in the AR leads to efficient access. Since the code already needs the ARP in a register, it can use ARP-relative offsets to access these values, with operations such as `loadAI` or `loadA0`. Frequent access to the AR will likely keep it in the data cache. The compiler places variables with either static lifetimes or global visibility into data areas in the "static" region of memory. Access to these values takes slightly more work at runtime; the compiler must ensure that it has an address for the data area in a register.

To establish the address of a static or global data area, the compiler typically loads a relocatable assembly language label.

Values stored in the heap have lifetimes that the compiler cannot easily predict. A value can be placed in the heap by two distinct mechanisms.

A PRIMER ON CACHE MEMORIES

One way that architects try to bridge the gap between processor speed and memory speed is through the use of *cache memories*. A cache is a small, fast memory placed between the processor and main memory. The cache is divided into a series of equal-sized *frames*. Each frame has an address field, called its *tag*, that holds a main-memory address.

The hardware automatically maps memory locations to cache frames. The simplest mapping, used in a direct-mapped cache, computes the cache address as the main memory address modulo the size of the cache. This partitions the memory into a linear set of blocks, each the size of a cache frame. A *line* is a memory block that maps to a frame. At any point in time, each cache frame holds a copy of the data from one of its blocks. Its tag field holds the address in memory where that data normally resides.

On each read access to memory, the hardware checks to see if the requested word is already in its cache frame. If so, the requested bytes are returned to the processor. If not, the block currently in the frame is evicted and the requested block is brought into the cache.

Some caches use more complex mappings. A set-associative cache uses multiple frames for each cache line, typically two or four frames per line. A fully associative cache can place any block in any frame. Both these schemes use an associative search over the tags to determine if a block is in the cache. Associative schemes use a policy to determine which block to evict; common schemes are random replacement and least-recently-used (LRU) replacement.

In practice, the effective memory speed is determined by memory bandwidth, cache block length, the ratio of cache speed to memory speed, and the percentage of accesses that hit in the cache. From the compiler's perspective, the first three are fixed. Compiler-based efforts to improve memory performance focus on increasing the ratio of cache hits to cache misses, called the hit ratio.

Some architectures provide instructions that allow a program to give the cache hints as to when specific blocks should be brought into memory (*prefetched*) and when they are no longer needed (*flushed*).

The programmer can explicitly allocate storage from the heap; the compiler should not override that decision. The compiler can place a value on the heap when it detects that the value might outlive the procedure that created it. In either case, a value in the heap is represented by a full address, rather than an offset from some base address.

Assigning Offsets

In the case of local, static, and global data areas, the compiler must assign each name an offset inside the data area. Target ISAs constrain the placement of data items in memory. A typical set of constraints might specify that 32-bit integers and 32-bit floating-point numbers begin on word (32-bit) boundaries, that 64-bit integer and floating-point data begin on doubleword (64-bit) boundaries, and that string data begin on halfword (16-bit) boundaries. We call these alignment rules.

Some processors provide operations to implement procedure calls beyond a simple jump operation. Such support often adds further alignment constraints. For example, the ISA might dictate the format of the AR and the alignment of the start of each AR. The DEC VAX computers had a particularly elaborate call instruction; it stored registers and other parts of the processor state based on a call-specific bit mask that the compiler produced.

For each data area, the compiler must compute a layout that assigns each variable in the data area its offset. That layout must comply with the ISA's alignment rules. The compiler may need to insert padding between some variables to obtain the proper alignments. To minimize wasted space, the compiler should order the variables into groups, from those with the most restrictive alignment rules to those with the least. (For example, doubleword alignment is more restrictive than word alignment.) The compiler then assigns offsets to the variables in the most restricted category, followed by the next most restricted class, and so on, until all variables have offsets. Since alignment rules almost always specify a power of two, the end of each category will naturally fit the restriction for the next category.

Most assembly languages have directives to specify the alignment of the start of a data area, such as a doubleword boundary.

Relative Offsets and Cache Performance

The widespread use of cache memories in modern computer systems has subtle implications for the layout of variables in memory. If two values are used in proximity in the code, the compiler would like to ensure that they can reside in the cache at the same time. This can be accomplished in two ways. In the best situation, the two values would share a single cache block, which guarantees that the values are fetched from memory to the cache together. If they cannot share a cache block, the compiler would like to ensure that the two variables map to different cache lines. The compiler can achieve this by controlling the distance between their addresses.

If we consider just two variables, controlling the distance between them seems manageable. When all the active variables are considered, however, the problem of optimal arrangement for a cache is NP-complete. Most

variables have interactions with many other variables; this creates a web of relationships that the compiler may not be able to satisfy concurrently. If we consider a loop that uses several large arrays, the problem of arranging mutual noninterference becomes even worse. If the compiler can discover the relationship between the various array references in the loop, it can add padding between the arrays to increase the likelihood that the references hit different cache lines and, thus, do not interfere with each other.

As we saw previously, the mapping of the program's logical address space to the hardware's physical address space need not preserve the distance between specific variables. Carrying this thought to its logical conclusion, the reader should ask how the compiler can ensure anything about relative offsets that are larger than the size of a virtual-memory page. The processor's cache may use either virtual addresses or physical addresses in its tag fields. A virtually addressed cache preserves the spacing between values that the compiler creates; with such a cache, the compiler may be able to plan noninterference between large objects. With a physically addressed cache, the distance between two locations in different pages is determined by the page mapping (unless cache size $\leq$ page size). Thus, the compiler's decisions about memory layout have little, if any, effect, except within a single page. In this situation, the compiler should focus on getting objects that are referenced together into the same page and, if possible, the same cache line.

7.2.3 Keeping Values in Registers

In a register-to-register memory model, the compiler tries to assign as many values as possible to virtual registers. In this approach, the compiler relies on the register allocator to map virtual registers in the IR to physical registers on the processor and to *spill* to memory any virtual register that it cannot keep in a physical register. If the compiler keeps a static value in a register, it must load the value before its first use in the procedure and store it back to memory before leaving the procedure, either at the procedure's exit or at any call site within the procedure.

Spill
When the register allocator cannot assign some virtual register to a physical register, it *spills* the value by storing it to RAM after each definition and loading it into a temporary register before each use.

In most of the examples in this book, we follow a simple method for assigning virtual registers to values. Each value receives its own virtual register with a distinct subscript. This discipline exposes the largest set of values to subsequent analysis and optimization. It may, in fact, use too many names. (See the digression, "The Impact of Naming" on page 248.) However, this scheme has three principal advantages. It is simple. It can improve the results of analysis and optimization. It prevents the compiler writer from

working processor-specific constraints into the code before optimization, thus enhancing portability. A strong register allocator can manage the name space and tailor it precisely to the needs of the application and the resources available on the target processor.

A value that the compiler can keep in a register is called an *unambiguous value*; a value that can have more than one name is called an *ambiguous value*. Ambiguity arises in several ways. Values stored in pointer-based variables are often ambiguous. Interactions between call-by-reference formal parameters and name scoping rules can make the formal parameters ambiguous. Many compilers treat array-element values as ambiguous values because the compiler cannot tell if two references, such as `A[i,j]` and `A[m,n]`, can ever refer to the same location. In general, the compiler cannot keep an ambiguous value in a register across either a definition or a use of another ambiguous value.

Unambiguous value
A value that can be accessed with just one name is *unambiguous*.

Ambiguous value
Any value that can be accessed by multiple names is *ambiguous*.

With careful analysis, the compiler can disambiguate some of these cases. Consider the sequence of assignments in the margin, assuming that both `a` and `b` are ambiguous. If `a` and `b` refer to the same location, then `c` gets the value 26; otherwise it receives `m+n+13`. The compiler cannot keep `a` in a register across an assignment to another ambiguous variable unless it can prove that the set of locations to which the two names can refer are disjoint. This kind of comparative pairwise analysis is expensive, so compilers typically relegate ambiguous values to memory, with a load before each use and a store after each definition.

```
a ← m + n;
b ← 13;
c ← a + b;
```

Analysis of ambiguity therefore focuses on proving that a given value is not ambiguous. The analysis might be cursory and local. For example, in C, any local variable whose address is never taken is unambiguous in the procedure where it is declared. More complex analyses build sets of possible names for each pointer variable; any variable whose set has just one element is unambiguous. Unfortunately, analysis cannot resolve all ambiguities. Thus, the compiler must be prepared to handle ambiguous values cautiously and correctly.

Language features can affect the compiler's ability to analyze ambiguity. For example, ANSI C includes two keywords that directly communicate information about ambiguity. The `restrict` keyword informs the compiler that a pointer is unambiguous. It is often used when a procedure passes an address directly at a call site. The `volatile` keyword lets the programmer declare that the contents of a variable may change arbitrarily and without notice. It is used for hardware device registers and for variables that might be modified by interrupt service routines or other threads of control in an application.

SECTION REVIEW

The compiler must determine, for each value computed in the program, where it must be stored: in memory or a register and, in either case, the specific location. It must assign to each value a location that is consistent with both its lifetime (see Section 6.3) and its addressability (see Section 6.4.3). Thus, the compiler will group together values into data areas in which each value has the same storage class.

Storage assignment provides the compiler with a key opportunity to encode information into the IR for use by later passes. Specifically, the distinction between an ambiguous value and an unambiguous value can be hard to derive by analysis of the IR. If, however, the compiler assigns each unambiguous value its own virtual register for its entire lifetime, subsequent phases of the compiler can use a value's storage location to determine whether or not a reference is ambiguous. This knowledge simplifies subsequent optimization.

```
void fee() {
  int a, *b;
    ...
  b = &a;
    ...
}
```

Review Questions

1. Sketch an algorithm that assigns offsets to a list of static variables in a single file from a C program. How does it order the variables? What alignment restrictions might your algorithm encounter?
2. Consider the short C fragment in the margin. It mentions three values: `a`, `b`, and `*b`. Which values are ambiguous? Which are unambiguous?

7.3 ARITHMETIC OPERATORS

Modern processors provide broad support for evaluating expressions. A typical RISC machine has a full complement of three-address operations, including arithmetic operators, shifts, and boolean operators. The three-address form lets the compiler name the result of any operation and preserve it for later reuse. It also eliminates the major complication of the two-address form: destructive operations.

To generate code for a trivial expression, such as `a+b`, the compiler first emits code to ensure that the values of `a` and `b` are in registers, say r_a and r_b. If `a` is stored in memory at offset `@a` in the current AR, the resulting code might be

```
loadI  @a       ⇒ r1
loadAO rarp,r1  ⇒ ra
```

If, however, the value of a is already in a register, the compiler can simply use that register in place of r_a. The compiler follows a similar chain of decisions for b. Finally, it emits an instruction to perform the addition, such as

```
add ra, rb ⇒ rt
```

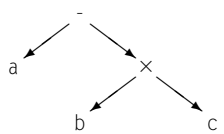
If the expression is represented in a tree-like IR, this process fits into a postorder tree walk. Figure 7.5a shows the code for a tree walk that generates code for simple expressions. It relies on two routines, *base* and *offset*, to hide some of the complexity. The *base* routine returns the name of a register holding the base address for an identifier; if needed, it emits code to get that address into a register. The *offset* routine has a similar function; it returns the name of a register holding the identifier's offset relative to the address returned by *base*.

```
expr(node) {
  int result, t1, t2;
  switch(type(node)) {
    case ×, ÷, +, -:
     t1 ← expr(LeftChild(node));
     t2 ← expr(RightChild(node));
     result ← NextRegister( );
     emit(op(node), t1, t2, result);
     break;

    case IDENT:
     t1 ← base(node);
     t2 ← offset(node);
     result ← NextRegister( );
     emit(loadAO, t1, t2, result);
     break;

    case NUM:
     result ← NextRegister( );
     emit(loadI, val(node), none,
          result);
     break;
  }
  return result;
}
```

(a) Treewalk Code Generator



(b) Abstract Syntax Tree for

a - b x c

```
loadI  @a       ⇒ r1
loadAO rarp, r1 ⇒ r2
loadI  @b       ⇒ r3
loadAO rarp, r3 ⇒ r4
loadI  @c       ⇒ r5
loadAO rarp, r5 ⇒ r6
mult   r4, r6   ⇒ r7
sub    r2, r7   ⇒ r8
```

(c) Naive Code

■ **FIGURE 7.5** Simple Treewalk Code Generator for Expressions.

The same code handles +, -, ×, and ÷. From a code-generation perspective, these operators are interchangeable, ignoring commutativity. Invoking the routine *expr* from Figure 7.5a on the AST for a - b x c shown in part b of the figure produces the results shown in part c of the figure. The example assumes that a, b, and c are not already in registers and that each resides in the current AR.

Notice the similarity between the treewalk code generator and the ad hoc syntax-directed translation scheme shown in Figure 4.15. The treewalk makes more details explicit, including the handling of terminals and the evaluation order for subtrees. In the syntax-directed translation scheme, the order of evaluation is controlled by the parser. Still, the two schemes produce roughly equivalent code.

7.3.1 Reducing Demand for Registers

Many issues affect the quality of the generated code. For example, the choice of storage locations has a direct impact, even for this simple expression. If a were in a global data area, the sequence of instructions needed to get a into a register might require an additional loadI to obtain the base address and a register to hold that value for a brief time. Alternatively, if a were in a register, the two instructions used to load it into r_2 could be omitted, and the compiler would use the name of the register holding a directly in the sub instruction. Keeping the value in a register avoids both the memory access and any address calculation. If a, b, and c were already in registers, the seven-instruction sequence could be shortened to a two-instruction sequence.

Code-shape decisions encoded into the treewalk code generator have an effect on demand for registers. The naive code in the figure uses eight registers, plus r_{arp}. It is tempting to assume that the register allocator, when it runs late in compilation, can reduce the number of registers to a minimum. For example, the register allocator could rewrite the code as shown in Figure 7.6a, which drops register use from eight registers to three, plus r_{arp}. The maximum demand for registers occurs in the sequence that loads c and performs the multiply.

A different code shape can reduce the demand for registers. The treewalk code generator loads a before it computes b x c, an artifact of the decision to use a left-to-right tree walk. Using a right-to-left tree walk would produce the code shown in Figure 7.6b. While the initial code uses the same number of registers as the code generated left-to-right, register allocation reveals that the code actually needs one fewer registers, as shown in Figure 7.6c.

```
loadI   @a       ⇒ r1
loadAO  rarp, r1 ⇒ r1
loadI   @b       ⇒ r2
loadAO  rarp, r2 ⇒ r2
loadI   @c       ⇒ r3
loadAO  rarp, r3 ⇒ r3
mult    r2, r3   ⇒ r2
sub     r1, r2   ⇒ r2
```

(a) Example After Allocation

```
loadI   @c       ⇒ r1
loadAO  rarp, r1 ⇒ r2
loadI   @b       ⇒ r3
loadAO  rarp, r3 ⇒ r4
mult    r2, r4   ⇒ r5
loadI   @a       ⇒ r6
loadAO  rarp, r6 ⇒ r7
sub     r7, r5   ⇒ r8
```

(b) Evaluating `b x c` First

```
loadI   @c       ⇒ r1
loadAO  rarp, r1 ⇒ r1
loadI   @b       ⇒ r2
loadAO  rarp, r2 ⇒ r2
mult    r1, r2   ⇒ r1
loadI   @a       ⇒ r2
loadAO  rarp, r2 ⇒ r2
sub     r2, r1   ⇒ r1
```

(c) After Register Allocation

■ **FIGURE 7.6** Rewriting `a - b x c` to Reduce Demand for Registers.

Of course, right-to-left evaluation is not a general solution. For the expression `a x b + c`, left-to-right evaluation produces the lower demand for registers. Some expressions, such as `a + (b + c) x d`, defy a simple static rule. The evaluation order that minimizes register demand is `a + ( (b + c) x d)`.

To choose an evaluation order that reduces demand for registers, the code generator must alternate between right and left children; it needs information about the detailed register needs of each subtree. As a rule, the compiler can minimize register use by evaluating first, at each node, the subtree that needs the most registers. The generated code must preserve the value of the first subtree that it evaluates across the evaluation of the second subtree; thus, handling the less demanding subtree first increases the demand for registers in the more demanding subtree by one register. This approach requires an initial pass over the code to compute demand for registers, followed by a pass that emits the actual code.

This approach, analysis followed by transformation, applies in both code generation and optimization [150].

7.3.2 Accessing Parameter Values

The code generator in Figure 7.5 implicitly assumes that a single access method works for all identifiers. Formal parameters may need different treatment. A call-by-value parameter passed in the AR can be handled as if it were a local variable. A call-by-reference parameter passed in the AR requires one additional indirection. Thus, for the call-by-reference parameter `d`, the compiler might generate

```
loadI   @d       ⇒ r1
loadAO  rarp, r1 ⇒ r2
load    r2       ⇒ r3
```

to obtain `d`'s value. The first two operations move the address of the parameter's value into r_2. The final operation moves the value itself into r_3.

GENERATING LOAD ADDRESS IMMEDIATE

A careful reader might notice that the code in Figure 7.5 never generates ILOC's load address-immediate instruction, `loadAI`. Instead, it generates a load immediate (`loadI`), followed by a load address-offset (`loadAO`):

```
loadI  @a        ⇒ r1
loadAO rarp, r1  ⇒ r2
```
instead of `loadAI rarp, @a ⇒ r2`

Throughout the book, the examples assume that it is preferable to generate this two-operation sequence, rather than the single operation. Three factors suggest this course.

1. The longer code sequence gives an explicit name to `@a`. If `@a` is reused in other contexts, that name can be reused.
2. The offset `@a` may not fit in the immediate field of a `loadAI`. That determination is best made in the instruction selector.
3. The two-operation sequence leads to a clean functional decomposition in the code generator, shown Figure 7.5.

The compiler can convert the two-operation sequence into a single operation during optimization, if appropriate (e.g. either `@a` is not reused or it is cheaper to reload it). The best course, however, may be to defer the issue to instruction selection, thus isolating the machine-dependent constant length into a part of the compiler that is already highly machine dependent.

If the compiler writer wants to generate the `loadAI` earlier, two simple approaches work. The compiler writer can refactor the treewalk code generator in Figure 7.5 and pull the logic hidden in *base* and *offset* into the case for *IDENT*. Alternatively, the compiler writer can have *emit* maintain a small instruction buffer, recognize this special case, and emit the `loadAI`. Using a small buffer makes this approach practical (see Section 11.5).

Many linkage conventions pass the first few parameters in registers. As written, the code in Figure 7.5 cannot handle a value that is permanently kept in a register. The necessary extensions, however, are easy to implement.

- *Call-by-value parameters* The *IDENT* case must check if the value is already in a register. If so, it just assigns the register number to *result*. Otherwise, it uses the standard mechanisms to load the value from memory.
- *Call-by-reference parameter* If the address resides in a register, the compiler simply loads the value into a register. If the address resides in the AR, it must load the address before it loads the value.

COMMUTATIVITY, ASSOCIATIVITY, AND NUMBER SYSTEMS

The compiler can often take advantage of algebraic properties of the operators. Addition and multiplication are commutative and associative, as are the boolean operators. Thus, if the compiler sees a code fragment that computes `a + b` and then computes `b + a`, with no intervening assignments to either `a` or `b`, it should recognize that they compute the same value. Similarly, if it sees the expressions `a + b + c` and `d + a + b`, it should recognize that `a + b` is a common subexpression. If it evaluates both expressions in strict left-to-right order, it will never recognize the common subexpression, since it will compute the second expression as `d + a` and then `( d + a ) + b`.

The compiler should use commutativity and associativity to improve the quality of code that it generates. Reordering expressions can expose additional opportunities for many transformations.

Due to limitations in precision, floating-point numbers on a computer represent only a subset of the real numbers, one that does not preserve associativity. For this reason, compilers should not reorder floating-point expressions unless the language definition specifically allows it.

Consider the following example: computing `a - b - c`. We can assign floating-point values to `a`, `b`, and `c` such that

$$b, c < a \qquad a - b = a \qquad a - c = a$$

but `a - ( b + c )` $\neq$ `a`. In that case, the numerical result depends on the order of evaluation. Evaluating `( a - b ) - c` produces a result identical to `a`, while evaluating `b + c` first and subtracting that quantity from `a` produces a result that is distinct from `a`.

This problem arises from the approximate nature of floating-point numbers; the mantissa is small relative to the range of the exponent. To add two numbers, the hardware must normalize them; if the difference in exponents is larger than the precision of the mantissa, the smaller number will be truncated to zero. The compiler cannot easily work its way around this issue, so it should, in general, avoid reordering float-point computations.

In either case, the code fits nicely into the treewalk framework. Note that the compiler cannot keep the value of a call-by-reference parameter in a register across an assignment, unless the compiler can prove that the reference is unambiguous, across all calls to the procedure.

If the actual parameter is a local variable of the caller and its address is never taken, the corresponding formal is unambiguous.

7.3.3 Function Calls in an Expression

So far, we have assumed that all the operands in an expression are variables, constants, and temporary values produced by other subexpressions. Function

calls also occur as operands in expressions. To evaluate a function call, the compiler simply generates the calling sequence needed to invoke the function and emits the code necessary to move the returned value to a register (see Section 7.9). The linkage convention limits the callee's impact on the caller.

The presence of a function call may restrict the compiler's ability to change an expression's evaluation order. The function may have side effects that modify the values of variables used in the expression. The compiler must respect the implied evaluation order of the source expression, at least with respect to the call. Without knowledge about the possible side effects of a call, the compiler cannot move references across the call. The compiler must assume the worst case—that the function both modifies and uses every variable that it can access. The desire to improve on worst-case assumptions, such as this one, has motivated much of the work in interprocedural analysis (see Section 9.4).

7.3.4 Other Arithmetic Operators

To handle other arithmetic operations, we can extend the treewalk model. The basic scheme remains the same: get the operands into registers, perform the operation, and store the result. Operator precedence, from the expression grammar, ensures the correct evaluation order. Some operators require complex multioperation sequences for their implementation (e.g. exponentiation and trigonometric functions). These may be expanded inline or implemented with a call to a library routine supplied by the compiler or the operating system.

7.3.5 Mixed-Type Expressions

One complication allowed by many programming languages is an operation with operands of different types. (Here, we are concerned primarily with base types in the source language, rather than programmer-defined types.) As described in Section 4.2, the compiler must recognize this situation and insert the conversion code required by each operator's conversion table. Typically, this involves converting one or both operands to a more general type and performing the operation in that more general type. The operation that consumes the result value may need to convert it to yet another type.

Some processors provide explicit conversion operators; others expect the compiler to generate complex, machine-dependent code. In either case, the compiler writer may want to provide conversion operators in the IR. Such an operator encapsulates all the details of the conversion, including any control flow, and lets the compiler subject it to uniform optimization. Thus, code

motion can pull an invariant conversion out of a loop without concern for the loop's internal control flow.

Typically, the programming-language definition specifies a formula for each conversion. For example, to convert `integer` to `complex` in FORTRAN 77, the compiler first converts the `integer` to a `real`. It uses the resulting number as the real part of the complex number and sets the imaginary part to a `real` zero.

For user-defined types, the compiler will not have conversion tables that define each specific case. However, the source language still defines the meaning of the expression. The compiler's task is to implement that meaning; if a conversion is illegal, then it should be prevented. As seen in Chapter 4, many illegal conversions can be detected and prevented at compile time. When a compile-time check is either impossible or inconclusive, the compiler should generate a runtime check that tests for illegal cases. When the code attempts an illegal conversion, the check should raise a runtime error.

7.3.6 Assignment as an Operator

Most Algol-like languages implement assignment with the following simple rules:

1. Evaluate the right-hand side of the assignment to a value.
2. Evaluate the left-hand side of the assignment to a location.
3. Store the right-hand side value into the left-hand side location.

Thus, in a statement such as `a` ← `b`, the two expressions `a` and `b` are evaluated differently. Since `b` appears to the right of the assignment operator, it is evaluated to produce a value; if `b` is an integer variable, that value is an integer. Since `a` is to the left of the assignment operator, it is evaluated to produce a location; if `a` is an integer variable, that value is the location of an integer. That location might be an address in memory, or it might be a register. To distinguish between these modes of evaluation, we sometimes refer to the result of evaluation on the right-hand side of an assignment as an *rvalue* and the result of evaluation on the left-hand side of an assignment as an *lvalue*.

Rvalue
An expression evaluated to a value is an *rvalue*.

Lvalue
An expression evaluated to a location is an *lvalue*.

In an assignment, the type of the lvalue can differ from the type of the rvalue. Depending on the language and the specific types, this situation may require either a compiler-inserted conversion or an error message. The typical source-language rule for conversion has the compiler evaluate the rvalue to its natural type and then convert the result to the type of the lvalue.

SECTION REVIEW

A postorder treewalk provides a natural way to structure a code generator for expression trees. The basic framework is easily adapted to handle a variety of complications, including multiple kinds and locations of values, function calls, type conversions, and new operators. To improve the code further may require multiple passes over the code.

Some optimizations are hard to fit into a treewalk framework. In particular, making good use of processor address modes (see Chapter 11), ordering operations to hide processor-specific delays (see Chapter 12), and register allocation (see Chapter 13) do not fit well into the treewalk framework. If the compiler uses a treewalk to generate IR, it may be best to keep the IR simple and allow the back end to address these issues with specialized algorithms.

Review Questions

1. Sketch the code for the two support routines, `base` and `offset`, used by the treewalk code generator in Figure 7.5.
2. How might you adapt the treewalk code generator to handle an unconditional jump operation, such as C's `goto` statement?

7.4 BOOLEAN AND RELATIONAL OPERATORS

Most programming languages operate on a richer set of values than numbers. Usually, this includes the results of boolean and relational operators, both of which produce boolean values. Because most programming languages have relational operators that produce boolean results, we treat the boolean and relational operators together. A common use for boolean and relational expressions is to alter the program's control flow. Much of the power of modern programming languages derives from the ability to compute and test such values.

Figure 7.7 shows the standard expression grammar augmented with boolean and relational operators. The compiler writer must, in turn, decide how to represent these values and how to compute them. For arithmetic expressions, such design decisions are largely dictated by the target architecture, which provides number formats and instructions to perform basic arithmetic. Fortunately, processor architects appear to have reached a widespread agreement about how to support arithmetic. Similarly, most architectures provide a rich set of boolean operations. However, support for relational operators varies widely from one architecture to another. The compiler writer must use an evaluation strategy that matches the needs of the language to the available instruction set.

The grammar uses the symbols ¬ for `not`, ∧ for `and`, and ∨ for `or` to avoid confusion with ILOC operators.

The type checker must ensure that each expression applies operators to names, numbers, and expressions of appropriate types.

Expr	→	*Expr* ∨ *AndTerm*	*NumExpr*	→	*NumExpr* + *Term*
	|	*AndTerm*		|	*NumExpr* − *Term*
AndTerm	→	*AndTerm* ∧ *RelExpr*		|	*Term*
	|	*RelExpr*	*Term*	→	*Term* × *Value*
RelExpr	→	*RelExpr* < *NumExpr*		|	*Term* ÷ *Value*
	|	*RelExpr* ≤ *NumExpr*		|	*Factor*
	|	*RelExpr* = *NumExpr*	*Value*	→	¬ *Factor*
	|	*RelExpr* ≠ *NumExpr*		|	*Factor*
	|	*RelExpr* ≥ *NumExpr*	*Factor*	→	(*Expr*)
	|	*RelExpr* > *NumExpr*		|	`num`
	|	*NumExpr*		|	`name`

■ **FIGURE 7.7** Adding Booleans and Relationals to the Expression Grammar.

7.4.1 Representations

Traditionally, two representations have been proposed for boolean values: a numerical encoding and a positional encoding. The former assigns specific values to true and false and manipulates them using the target machine's arithmetic and logical operations. The latter approach encodes the value of the expression as a position in the executable code. It uses comparisons and conditional branches to evaluate the expression; the different control-flow paths represent the result of evaluation. Each approach works well for some examples, but not for others.

Numerical Encoding

When the program stores the result of a boolean or relational operation into a variable, the compiler must ensure that the value has a concrete representation. The compiler writer must assign numerical values to true and false that work with the hardware operations such as `and`, `or`, and `not`. Typical values are zero for false and either one or a word of ones, ¬`false`, for true.

For example, if `b`, `c`, and `d` are all in registers, the compiler might produce the following code for the expression `b` ∨ `c` ∧ ¬ `d`:

```
not r_d      ⇒ r_1
and r_c, r_1 ⇒ r_2
or  r_b, r_2 ⇒ r_3
```

For a comparison, such as `a` < `b`, the compiler must generate code that compares `a` and `b` and assigns the appropriate value to the result. If the target machine supports a comparison operation that returns a boolean, the code is trivial:

```
cmp_LT r_a, r_b ⇒ r_1
```

ILOC contains syntax to implement both styles of compare and branch. A normal IR would choose one; ILOC includes both so that it can express the code in this section.

If, on the other hand, the comparison defines a condition code that must be read with a branch, the resulting code is longer and more involved. This style of comparison leads to a messier implementation for a < b.

```
    comp   ra, rb ⇒ cc1
    cbr_LT cc1    → L1, L2
L1: loadI  true   ⇒ r1
    jumpI         → L3
L2: loadI  false  ⇒ r1
    jumpI         → L3
L3: nop
```

Implementing a < b with condition-code operations requires more operations than using a comparison that returns a boolean.

Positional Encoding

In the previous example, the code at L_1 creates the value true and the code at L_2 creates the value false. At each of those points, the value is known. In some cases, the code need not produce a concrete value for the expression's result. Instead, the compiler can encode that value in a location in the code, such as L_1 or L_2.

Figure 7.8a shows the code that a treewalk code generator might emit for the expression a < b ∨ c < d ∧ e < f. The code evaluates the three subexpressions, a < b, c < d, and e < f, using a series of comparisons and jumps. It then combines the result of the three subexpression evaluations using the boolean operations at L_9. Unfortunately, this produces a sequence of operations in which every path takes 11 operations, including three branches and three jumps. Some of the complexity of this code can be eliminated by representing the subexpression values implicitly and generating code that short circuits the evaluation, as in Figure 7.8b. This version of the code evaluates a < b ∨ c < d ∧ e < f with fewer operations because it does not create values to represent the subexpressions.

Positional encoding makes sense if an expression's result is never stored. When the code uses the result of an expression to determine control flow, positional encoding often avoids extraneous operations. For example, in the code fragment

```
if (a < b)
   then statement1
   else statement2
```

the sole use for a < b is to determine whether $statement_1$ or $statement_2$ executes. Producing an explicit value for a < b serves no direct purpose.

```
     comp    ra, rb  ⇒ cc1     // a < b
     cbr_LT  cc1     → L1, L2
L1:  loadI   true    ⇒ r1
     jumpI   → L3
L2:  loadI   false   ⇒ r1
     jumpI   → L3
L3:  comp    rc, rd  ⇒ cc2     // c < d
     cbr_LT  cc2     → L4, L5
L4:  loadI   true    ⇒ r2
     jumpI   → L6
L5:  loadI   false   ⇒ r2
     jumpI   → L6
L6:  comp    re, rf  ⇒ cc3     // e < f
     cbr_LT  cc3     → L7, L8
L7:  loadI   true    ⇒ r3
     jumpI           → L9
L8:  loadI   false   ⇒ r3
     jumpI           → L9
L9:  and     r2, r3  ⇒ r4
     or      r1, r4  ⇒ r5
```

(a) Naive Encoding

```
     comp    ra, rb  ⇒ cc1     // a < b
     cbr_LT  cc1     → L3, L1
L1:  comp    rc, rd  ⇒ cc2     // c < d
     cbr_LT  cc2     → L2, L4
L2:  comp    re, rf  ⇒ cc3     // e < f
     cbr_LT  cc3     → L3, L4
L3:  loadI   true    ⇒ r5
     jumpI           → L5
L4:  loadI   false   ⇒ r5
     jumpI           → L5
L5:  nop
```

(b) Positional Encoding with Short-Circuit Evaluation

■ **FIGURE 7.8** Encoding a < b ∨ c < d ∧ e < f.

On a machine where the compiler must use a comparison and a branch to produce a value, the compiler can simply place the code for *statement*$_1$ and *statement*$_2$ in the locations where naive code would assign true and false. This use of positional encoding leads to simpler, faster code than using numerical encoding.

```
     comp    ra, rb  ⇒ cc1     // a < b
     cbr_LT  cc1     → L1, L2
L1:  code for statement1
     jumpI           → L6
L2:  code for statement2
     jumpI           → L6
L6:  nop
```

Here, the code to evaluate a < b has been combined with the code to select between *statement*$_1$ and *statement*$_2$. The code represents the result of a < b as a position, either L_1 or L_2.

7.4.2 Hardware Support for Relational Operations

Specific, low-level details in the target machine's instruction set strongly influence the choice of a representation for relational values. In particular,

SHORT-CIRCUIT EVALUATION

In many cases, the value of a subexpression determines the value of the entire expression. For example, the code shown in Figure 7.8a, evaluates `c < d ∧ e < f`, even if it has already determined that `a < b`, in which case the entire expression evaluates to true. Similarly, if both `a ≥ b` and `c ≥ d`, then the value of `e < f` does not matter. The code in Figure 7.8b uses these relationships to produce a result as soon as the expression's value can be known. This approach to expression evaluation, in which the code evaluates the minimal amount of the expression needed to determine its final value, is called *short-circuit evaluation*. Short-circuit evaluation relies on two boolean identities:

```
∀x, false ∧ x = false
∀x,  true ∨ x =  true
```

To generate the short-circuit code, the compiler must analyze the expression in light of these two identities and find the set of minimal conditions that determine its value. If clauses in the expression contain expensive operators or if the evaluation uses branches, as do many of the schemes discussed in this section, then short-circuit evaluation can significantly reduce the cost of evaluating boolean expressions.

Some programming languages, like C, require the compiler to use short-circuit evaluation. For example, the expression

```
(x != 0 && y/x > 0.001)
```

in C relies on short-circuit evaluation for safety. If `x` is zero, `y / x` is not defined. Clearly, the programmer intends to avoid the hardware exception triggered by division by zero. The language definition specifies that this code will never perform the division if `x` has the value zero.

the compiler writer must pay attention to the handling of condition codes, compare operations, and conditional move operations, as they have a major impact on the relative costs of the various representations. We will consider four schemes for supporting relational expressions: straight condition codes, condition codes augmented with a conditional move operation, boolean-valued comparisons, and predicated operations. Each scheme is an idealized version of a real implementation.

Figure 7.9 shows two source-level constructs and their implementations under each of these schemes. Figure 7.9a shows an `if-then-else` that controls a pair of assignment statements. Figure 7.9b shows the assignment of a boolean value.

Source Code

```
if (x < y)
   then a ← c + d
   else a ← e + f
```

ILOC Code

Straight Condition Codes

```
     comp   rx, ry ⇒ cc1
     cbr_LT cc1    → L1, L2
L1:  add    rc, rd ⇒ ra
     jumpI         → Lout
L2:  add    re, rf ⇒ ra
     jumpI         → Lout
Lout: nop
```

Boolean Compare

```
     cmp_LT rx, ry ⇒ r1
     cbr    r1     → L1, L2
L1:  add    rc, rd ⇒ ra
     jumpI         → Lout
L2:  add    re, rf ⇒ ra
     jumpI         → Lout
Lout: nop
```

Conditional Move

```
comp   rx, ry      ⇒ cc1
add    rc, rd      ⇒ r1
add    re, rf      ⇒ r2
i2i_LT cc1, r1, r2 ⇒ ra
```

Predicated Execution

```
     cmp_LT rx, ry ⇒ r1
     not    r1     ⇒ r2
(r1)? add   rc, rd ⇒ ra
(r2)? add   re, rf ⇒ ra
```

(a) Using a Relational Expression to Govern Control Flow

Source Code

```
x ← a < b ∧ c < d
```

ILOC Code

Straight Condition Codes

```
     comp   ra, rb ⇒ cc1
     cbr_LT cc1    → L1, L2
L1:  comp   rc, rd ⇒ cc2
     cbr_LT cc2    → L3, L2
L2:  loadI  false  ⇒ rx
     jumpI         → Lout
L3:  loadI  true   ⇒ rx
     jumpI         → Lout
Lout: nop
```

Conditional Move

```
comp   ra, rb      ⇒ cc1
i2i_LT cc1, rT, rF ⇒ r1
comp   rc, rd      ⇒ cc2
i2i_LT cc2, rT, rF ⇒ r2
and    r1, r2      ⇒ rx
```

Boolean Compare

```
cmp_LT ra, rb ⇒ r1
cmp_LT rc, rd ⇒ r2
and    r1, r2 ⇒ rx
```

Predicated Execution

```
cmp_LT ra, rb ⇒ r1
cmp_LT rc, rd ⇒ r2
and    r1, r2 ⇒ rx
```

(b) Using a Relational Expression to Produce a Value

■ **FIGURE 7.9** Implementing Boolean and Relational Operators.

Straight Condition Codes

In this scheme, the comparison operation sets a condition-code register. The only instruction that interprets the condition code is a conditional branch, with variants that branch on each of the six relations ($<$, $\leq$, $=$, $\geq$, $>$, and $\neq$). These instructions may exist for operands of several types.

SHORT-CIRCUIT EVALUATION AS AN OPTIMIZATION

Short-circuit evaluation arose from a positional encoding of the values of boolean and relational expressions. On processors that use condition codes to record the result of a comparison and use conditional branches to interpret the condition code, short circuiting makes sense.

As processors include features like conditional move, boolean-valued comparisons, and predicated execution, the advantages of short-circuit evaluation will likely fade. With branch latencies growing, the cost of the conditional branches required for short circuiting grows too. When the branch costs exceed the savings from avoiding evaluation, short circuiting will no longer be an improvement. Instead, full evaluation will be faster.

When the language requires short-circuit evaluation, as does C, the compiler may need to perform some analysis to determine when it is safe to substitute full evaluation for short-circuit evaluation. Thus, future C compilers may include analysis and transformation to replace short circuiting with full evaluation, just as compilers in the past have performed analysis and transformation to replace full evaluation with short-circuit evaluation.

The compiler must use conditional branches to interpret the value of a condition code. If the sole use of the result is to determine control flow, as in Figure 7.9a, then the conditional branch that the compiler uses to read the condition code can often implement the source-level control-flow construct, as well. If the result is used in a boolean operation, or it is preserved in a variable, as in Figure 7.9b, the code must convert the result into a concrete representation of a boolean, as do the two `loadI` operations in Figure 7.9b. Either way, the code has at least one conditional branch per relational operator.

The advantage of condition codes comes from another feature that processors usually implement alongside condition codes. Typically, arithmetic operations on these processors set the condition code to reflect their computed results. If the compiler can arrange to have the arithmetic operations that must be performed also set the condition code needed to control the branch, then the comparison operation can be omitted. Thus, advocates of this architectural style argue that it allows a more efficient encoding of the program—the code may execute fewer instructions than it would with a comparator that puts a boolean value in a general-purpose register.

Conditional Move

This scheme adds a conditional move instruction to the straight condition-code model. In ILOC, a conditional move looks like:

$$\texttt{i2i_LT}\ cc_i, r_j, r_k \Rightarrow r_m$$

If the condition code cc_i matches LT, then the value of r_j is copied to r_m. Otherwise, the value of r_k is copied to r_m. The conditional move operation typically executes in a single cycle. It leads to faster code by allowing the compiler to avoid branches.

Conditional move retains the principal advantage of using condition codes—avoiding a comparison when an earlier operation has already set the condition code. As shown in Figure 7.9a, it lets the compiler encode simple conditional operations with branches. Here, the compiler speculatively evaluates the two additions. It uses conditional move for the final assignment. This is safe as long as neither addition can raise an exception.

If the compiler has values for true and false in registers, say r_T for true and r_F for false, then it can use conditional move to convert the condition code into a boolean. Figure 7.9b uses this strategy. It compares a and b and places the boolean result in r_1. It computes the boolean for c < d into r_2. It computes the final result as the logical and of r_1 and r_2.

Boolean-Valued Comparisons

This scheme avoids condition codes entirely. The comparison operator returns a boolean value in a register. The conditional branch takes that result as an argument that determines its behavior.

Boolean-valued comparisons do not help with the code in Figure 7.9a. The code is equivalent to the straight condition-code scheme. It requires comparisons, branches, and jumps to evaluate the if-then-else construct.

Figure 7.9b shows the strength of this scheme. The boolean compare lets the code evaluate the relational operator without a branch and without converting comparison results to boolean values. The uniform representation of boolean and relational values leads to concise, efficient code for this example.

A weakness of this model is that it requires explicit comparisons. Whereas the condition-code models can sometimes avoid the comparison by arranging to set the appropriate condition code with an earlier arithmetic operation, the boolean-valued comparison model always needs an explicit comparison.

Predicated Execution

Predicated execution
an architectural feature in which some operations take a boolean-valued operand that determines whether or not the operation takes effect

Architectures that support *predicated execution* let the compiler avoid some conditional branches. In ILOC, we write a predicated instruction by including a predicate expression before the instruction. To remind the reader of

the predicate's purpose, we enclose it in parentheses and follow it with a question mark. For example,

$$(r_{17})?\ \texttt{add}\ r_a, r_b \Rightarrow r_c$$

indicates an add operation ($r_a + r_b$) that executes if and only if r_{17} contains true.

The example in Figure 7.9a shows the strength of predicated execution. The code is simple and concise. It generates two predicates, r_1 and r_2. It uses them to control the code in the `then` and `else` parts of the source construct. In Figure 7.9b, predication leads to the same code as the boolean-comparison scheme.

The processor can use predication to avoid executing the operation, or it can execute the operation and use the predicate to avoid assigning the result. As long as the idled operation does not raise an exception, the differences between these two approaches are irrelevant to our discussion. Our examples show the operations required to produce both the predicate and its complement. To avoid the extra computation, a processor could provide comparisons that return two values, both the boolean value and its complement.

SECTION REVIEW

The implementation of boolean and relational operators involves more choices than the implementation of arithmetic operators. The compiler writer must choose between a numerical encoding and a positional encoding. The compiler must map those decisions onto the set of operations provided by the target processor's ISA.

In practice, compilers choose between numerical and positional encoding based on context. If the code instantiates the value, numerical encoding is necessary. If the value's only use is to determine control flow, positional encoding often produces better results.

Review Questions

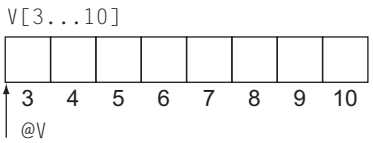
1. If the compiler assigns the value zero to false, what are the relative merits of each of the following values for true? One? Any non-zero number? A word composed entirely of ones?
2. How might the treewalk code generation scheme be adapted to generate positional code for boolean and relational expressions? Can you work short-circuit evaluation into your approach?

7.5 STORING AND ACCESSING ARRAYS

So far, we have assumed that variables stored in memory contain scalar values. Many programs need arrays or similar structures. The code required to locate and reference an element of an array is surprisingly complex. This section shows several schemes for laying out arrays in memory and describes the code that each scheme produces for an array reference.

7.5.1 Referencing a Vector Element

The simplest form of an array has a single dimension; we call it a *vector*. Vectors are typically stored in contiguous memory, so that the i^{th} element immediately precedes the $i{+}1^{st}$ element. Thus, a vector `V[3...10]` generates the following memory layout, where the number below a cell indicates its index in the vector:



When the compiler encounters a reference, like `V[6]`, it must use the index into the vector, along with facts available from the declaration of `V`, to generate an offset for `V[6]`. The actual address is then computed as the sum of the offset and a pointer to the start of `V`, which we write as `@V`.

As an example, assume that `V` has been declared as `V[`*low*`...`*high*`]`, where *low* and *high* are the vector's lower and upper bounds. To translate the reference `V[i]`, the compiler needs both a pointer to the start of storage for `V` and the offset of element `i` within `V`. The offset is simply (`i` − *low*) × *w*, where *w* is the length of a single element of `V`. Thus, if *low* is 3, `i` is 6, and *w* is 4, the offset is $(6-3)\times 4 = 12$. Assuming that r_i holds the value of `i`, the following code fragment computes the address of `V[i]` into r_3 and loads its value into r_V:

```
loadI @V      ⇒ r@V  // get V's address
subI  ri,3    ⇒ r1   // (offset - lower bound)
multI r1,4    ⇒ r2   // x element length (4)
add   r@V,r2  ⇒ r3   // address of V[i]
load  r3      ⇒ rV   // value of V[i]
```

Notice that the simple reference `V[i]` introduces three arithmetic operations. The compiler can improve this sequence. If *w* is a power of two, the multiply

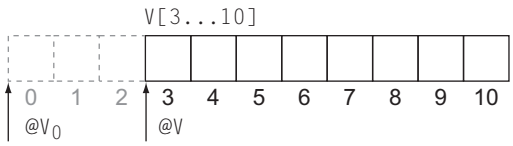
can be replaced with an arithmetic shift; many base types in real programming languages have this property. Adding the address and offset seems unavoidable; perhaps this explains why most processors include an addressing mode that takes a base address and an offset and accesses the location at base address + offset. In ILOC, we write this as `loadAO`.

```
loadI   @V      ⇒ r@V  // get V's address
subI    ri, 3   ⇒ r1   // (offset - lower bound)
lshiftI r1, 2   ⇒ r2   // x element length (4)
loadAO  r@V, r2 ⇒ rV   // value of V[i]
```

False zero
The false zero of a vector V is the address where V[0] would be.

In multiple dimensions, it is the location of a zero in each dimension.

Using a lower bound of zero eliminates the subtraction. If the compiler knows the lower bound of `V`, it can fold the subtraction into `@V`. Rather than using `@V` as the base address for `V`, it can use $\texttt{@V}_0 = \texttt{@V} - low \times w$. We call $\texttt{@V}_0$ the *false zero* of `V`.



Using $\texttt{@V}_0$ and assuming that `i` is in $\texttt{r}_i$, the code for accessing `V[i]` becomes

```
loadI   @V0      ⇒ r@V0  // adjusted address for V
lshiftI ri, 2    ⇒ r1    // x element length (4)
loadAO  r@V0, r1 ⇒ rV    // value of V[i]
```

This code is shorter and, presumably, faster. A good assembly-language programmer might write this code. In a compiler, the longer sequence may produce better results by exposing details such as the multiply and add to optimization. Low-level improvements, such as converting the multiply into a shift and converting the `add`–`load` sequence into with `loadAO`, can be done late in compilation.

If the compiler does not know an array's bounds, it might calculate the array's false zero at runtime and reuse that value in each reference to the array. It might compute the false zero on entry to a procedure that references elements of the array multiple times. An alternative strategy, employed in languages like C, forces the use of zero as a lower bound, which ensures that $\texttt{@V}_0 = \texttt{@V}$ and simplifies all array-address calculations. However, attention to detail in the compiler can achieve the same results without restricting the programmer's choice of a lower bound.

7.5.2 Array Storage Layout

Accessing an element of a multidimensional array requires more work. Before discussing the code sequences that the compiler must generate, we must consider how the compiler will map array indices to memory locations. Most implementations use one of three schemes: *row-major order, column-major order*, or *indirection vectors*. The source-language definition usually specifies one of these mappings.

The code required to access an array element depends on the way that the array is mapped to memory. Consider the array `A[1...2,1...4]`. Conceptually, it looks like

A

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4

In linear algebra, the *row* of a two-dimensional matrix is its first dimension, and the *column* is its second dimension. In row-major order, the elements of `a` are mapped onto consecutive memory locations so that adjacent elements of a single row occupy consecutive memory locations. This produces the following layout:

1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4

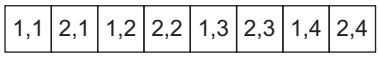
The following loop nest shows the effect of row-major order on memory access patterns:

```
for i ← 1 to 2
   for j ← 1 to 4
      A[i,j] ← A[i,j] + 1
```

In row-major order, the assignment statement steps through memory in sequential order, beginning with `A[1,1]`, `A[1,2]`, `A[1,3]`, and on through `A[2,4]`. This sequential access works well with most memory hierarchies. Moving the `i` loop inside the `j` loop produces an access sequence that jumps between rows, accessing `A[1,1]`, `A[2,1]`, `A[1,2]`, ..., `A[2,4]`. For a small array like `a`, this is not a problem. For arrays that are larger than the cache, the lack of sequential access could produce poor performance in the memory hierarchy. As a general rule, row-major order produces sequential access when the rightmost subscript, `j` in this example, varies fastest.

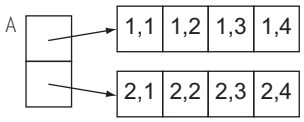
FORTRAN uses column-major order.

The obvious alternative to row-major order is column-major order. It keeps the columns of `a` in contiguous locations, producing the following layout:



Column-major order produces sequential access when the leftmost subscript varies fastest. In our doubly nested loop, having the `i` loop in the outer position produces nonsequential access, while moving the `i` loop to the inner position would produce sequential access.

A third alternative, not quite as obvious, has been used in several languages. This scheme uses indirection vectors to reduce all multidimensional arrays to a set of vectors. For our array `a`, this would produce



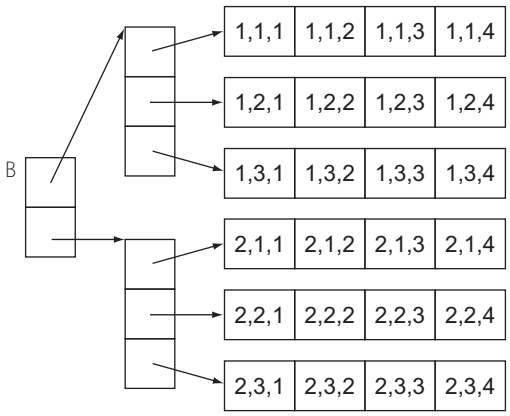
Each row has its own contiguous storage. Within a row, elements are addressed as in a vector. To allow systematic addressing of the row vectors, the compiler allocates a vector of pointers and initializes it appropriately. A similar scheme can create column-major indirection vectors.

Indirection vectors appear simple, but they introduce their own complexity. First, indirection vectors require more storage than either of the contiguous storage schemes, as shown graphically in Figure 7.10. Second, this scheme requires that the application initialize, at runtime, all of the indirection pointers. An advantage of the indirection vector approach is that it allows easy implementation of ragged arrays, that is, arrays where the length of the last dimension varies.

Each of these schemes has been used in a popular programming language. For languages that store arrays in contiguous storage, row-major order has been the typical choice; the one notable exception is FORTRAN, which uses column-major order. Both BCPL and Java support indirection vectors.

7.5.3 Referencing an Array Element

Programs that use arrays typically contain references to individual array elements. As with vectors, the compiler must translate an array reference into a base address for the array's storage and an offset where the element is located relative to the starting address.



■ **FIGURE 7.10** Indirection Vectors in Row-Major Order for `B[1...2,1...3,1...4]`.

This section describes the address calculations for arrays stored as a contiguous block in row-major order and as a set of indirection vectors. The calculations for column-major order follow the same basic scheme as those for row-major order, with the dimensions reversed. We leave those equations for the reader to derive.

Row-Major Order

In row-major order, the address calculation must find the start of the row and then generate an offset within the row as if it were a vector. Extending the notation that we used to describe the bounds of a vector, we add subscripts to *low* and *high* that specify a dimension. Thus, low_1 refers to the lower bound of the first dimension, and $high_2$ refers to the upper bound of the second dimension. In our example `A[1...2,1...4]`, low_1 is 1 and $high_2$ is 4.

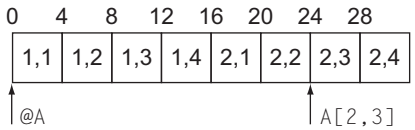
To access element `A[i,j]`, the compiler must emit code that computes the address of row `i` and follow that with the offset for element `j`, which we know from Section 7.5.1 will be $(\texttt{j} - low_2) \times w$. Each row contains four elements, computed as $high_2 - low_2 + 1$, where $high_2$ is the highest-numbered column and low_2 is the lowest-numbered column—the upper and lower bounds for the second dimension of `A`. To simplify the exposition, let $len_k = high_k - low_k + 1$, the length of the k^{th} dimension. Since rows are laid out consecutively, row `i` begins at $(\texttt{i} - low_1) \times len_2 \times w$ from the start of `A`. This suggests the address computation

$$@\texttt{A} + (\texttt{i} - low_1) \times len_2 \times w + (\texttt{j} - low_2) \times w$$

Substituting actual values for i, j, low_1, $high_2$, low_2, and w, we find that A[2,3] lies at offset

$$(2 - 1) \times (4 - 1 + 1) \times 4 + (3 - 1) \times 4 = 2$$

from A[1,1] (assuming that @A points at A[1,1], at offset 0). Looking at A in memory, we find that the address of A[1,1] + 24 is, in fact, the address of A[2,3].



In the vector case, we were able to simplify the calculation when upper and lower bounds were known at compile time. Applying the same algebra to create a false zero in the two-dimensional case produces

$$@A + (i \times len_2 \times w) - (low_1 \times len_2 \times w) + (j \times w) - (low_2 \times w), \text{ or}$$

$$@A + (i \times len_2 \times w) + (j \times w) - (low_1 \times len_2 \times w + low_2 \times w)$$

The last term, $(low_1 \times len_2 \times w + low_2 \times w)$, is independent of i and j, so it can be factored directly into the base address

$$@A_0 = @A - (low_1 \times len_2 \times w + low_2 \times w) = @A - 20$$

Now, the array reference is simply

$$@A_0 + i \times len_2 \times w + j \times w$$

Finally, we can refactor and move the w outside, saving an extraneous multiply

$$@A_0 + (i \times len_2 + j) \times w$$

For the address of A[2,3], this evaluates to

$$@A_0 + (2 \times 4 + 3) \times 4 = @A_0 + 44$$

Since $@A_0$ is just $@A - 20$, this is equivalent to $@A - 20 + 44 = @A + 24$, the same location found with the original version of the array address polynomial.

If we assume that i and j are in r_i and r_j, and that len_2 is a constant, this form of the polynomial leads to the following code sequence:

```
loadI  @A0      ⇒ r@A0  // adjusted base for A
multI  ri, len2 ⇒ r1    // i × len2
add    r1, rj   ⇒ r2    // + j
multI  r2, 4    ⇒ r3    // x element length, 4
loadAO r@A0, r3 ⇒ ra    // value of A[i,j]
```

In this form, we have reduced the computation to two multiplications and two additions (one in the loadAO). The second multiply can be rewritten as a shift.

If the compiler does not have access to the array bounds, it must either compute the false zero at runtime or use the more complex polynomial that includes the subtractions that adjust for lower bounds. The former option can be profitable if the elements of the array are accessed multiple times in a procedure; computing the false zero on entry to the procedure lets the code use the less expensive address computation. The more complex computation makes sense only if the array is accessed infrequently.

The ideas behind the address computation for arrays with two dimensions generalize to arrays of higher dimension. The address polynomial for an array stored in column-major order can be derived in a similar fashion. The optimizations that we applied to reduce the cost of address computations apply equally well to the address polynomials for these other kinds of arrays.

Indirection Vectors

Using indirection vectors simplifies the code generated to access an individual element. Since the outermost dimension is stored as a set of vectors, the final step looks like the vector access described in Section 7.5.1. For B[i,j,k], the final step computes an offset from k, the outermost dimension's lower bound, and the length of an element for B. The preliminary steps derive the starting address for this vector by following the appropriate pointers through the indirection-vector structure.

Thus, to access element B[i,j,k] in the array B shown in Figure 7.10, the compiler uses $@B_0$, i, and the length of a pointer, to find the vector for the subarray B[i,*,*]. Next, it uses that result, along with j and the length of a pointer to find the vector for the subarray B[i,j,*]. Finally, it uses that base address in the vector-address computation with k and element length w to find the address of B[i,j,k].

If the current values for i, j, and k exist in registers r_i, r_j, and r_k, respectively, and $@B_0$ is the zero-adjusted address of the first dimension, then B[i,j,k] can be referenced as follows:

```
loadI  @B0      ⇒ r@B0   // false zero of B
multI  ri,4     ⇒ r1     // assume pointer is 4 bytes
loadAO r@B0,r1  ⇒ r2     // get @B[i,*,*]
multI  rj,4     ⇒ r3     // pointer is 4 bytes
loadAO r2,r3    ⇒ r4     // get @B[i,j,*]
multI  rk,4     ⇒ r5     // assume element length is 4
loadAO r4,r5    ⇒ rb     // value of B[i,j,k]
```

This code assumes that the pointers in the indirection structure have already been adjusted to account for nonzero lower bounds. If that is not the case, then the values in r_j and r_k must be decremented by the corresponding lower bounds. The multiplies can be replaced by shifts in this example.

Using indirection vectors, the reference requires just two operations per dimension. This property made the indirection-vector scheme efficient on systems in which memory access is fast relative to arithmetic—for example, on most computer systems prior to 1985. As the cost of memory accesses has increased relative to arithmetic, this scheme has lost its advantage in speed.

On cache-based machines, locality is critical to performance. When arrays grow to be much larger than the cache, storage order affects locality. Row-major and column-major storage schemes produce good locality for some array-based operations. The locality properties of an array implemented with indirection vectors are harder for the compiler to predict and, perhaps, to optimize.

Accessing Array-Valued Parameters

When an array is passed as a parameter, most implementations pass it by reference. Even in languages that use call by value for all other parameters, arrays are usually passed by reference. Consider the mechanism required to pass an array by value. The caller would need to copy each array element's value into the activation record of the callee. Passing the array as a reference parameter can greatly reduce the cost of each call.

If the compiler is to generate array references in the callee, it needs information about the dimensions of the array that is bound to the parameter. In FORTRAN, for example, the programmer is required to declare the array using either constants or other formal parameters to specify its dimensions. Thus, FORTRAN gives the programmer responsibility for passing to the callee the information that it needs to address correctly a parameter array.

Other languages leave the task of collecting, organizing, and passing the necessary information to the compiler. The compiler builds a descriptor that contains both a pointer to the start of the array and the necessary information for each dimension. The descriptor has a known size, even when the array's size cannot be known at compile time. Thus, the compiler can allocate space for the descriptor in the AR of the callee procedure. The value passed in the array's parameter slot is a pointer to this descriptor, which is called a *dope vector*.

Dope vector
a descriptor for an actual parameter array

Dope vectors may also be used for arrays whose bounds are determined at runtime.

When the compiler generates a reference to a formal-parameter array, it must extract the information from the dope vector. It generates the same address polynomial that it would use for a reference to a local array, loading values out of the dope vector as needed. The compiler must decide, as a matter of policy, which form of the address polynomial it will use. With the naive address polynomial, the dope vector contains a pointer to the start of the array, the lower bound of each dimension, and the sizes of all but one of the dimensions. With the address polynomial based on the false zero, the lower-bound information is unneeded. Because it may compile caller and callee separately, the compiler must be consistent in its usage. In most cases, the code to build the actual dope vector can be moved away from the call site and placed in the caller's prologue code. For a call inside a loop, this move reduces the call overhead.

One procedure might be invoked from multiple call sites, each passing a different array. The PL/I procedure `main` in Figure 7.11a contains two calls to procedure `fee`. The first passes the array `x`, while the second passes `y`. Inside `fee`, the actual parameter (`x` or `y`) is bound to the formal parameter `A`. The code in `fee` for a reference to `A` needs a dope vector to describe the actual parameter. Figure 7.11b shows the respective dope vectors for the two call sites, based on the false-zero version of the address polynomial.

Notice that the cost of accessing an array-valued parameter or a dynamically sized array is higher than the cost of accessing a local array with fixed bounds. At best, the dope vector introduces additional memory references to access the relevant entries. At worst, it prevents the compiler from performing optimizations that rely on complete knowledge of an array's declaration.

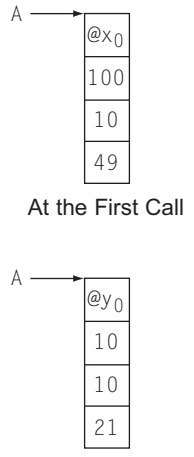
7.5.4 Range Checking

Most programming-language definitions assume, either explicitly or implicitly, that a program refers only to array elements within the defined bounds of an array. A program that references an out-of-bounds element is, by definition, not well formed. Some languages (for example, Java and Ada) require that out-of-bounds accesses be detected and reported. In other

```
program main;
  begin;
    declare x(1:100,1:10,2:50),
        y(1:10,1:10,15:35) float;
    ...
    call fee(x)
    call fee(y);
  end main;

procedure fee(A)
  declare A(*,*,*) float;
  begin;
    declare x float;
      declare i, j, k fixed binary;
      ...
      x = A(i,j,k);
      ...
  end fee;
```

(a) Code that Passes Whole Arrays



(b) Dope Vectors for the Call Sites

■ **FIGURE 7.11** Dope Vectors.

languages, compilers have included optional mechanisms to detect and report out-of-bounds array accesses.

The simplest implementation of *range checking*, as this is called, inserts a test before each array reference. The test verifies that each index value falls in the valid range for the dimension in which it is used. In an array-intensive program, the overhead of such checks can be significant. Many improvements on this simple scheme are possible. The least expensive alternative is to prove, in the compiler, that a given reference cannot generate an out-of-bounds reference.

If the compiler intends to insert range checks for array-valued parameters, it may need to include additional information in the dope vectors. For example, if the compiler uses the address polynomial based on the array's false zero, it has length information for each dimension, but not upper and lower bound information. It might perform an imprecise test by checking the offset against the array's overall length. However, to perform a precise test, the compiler must include the upper and lower bounds for each dimension in the dope vector and test against them.

When the compiler generates runtime code for range checking, it inserts many copies of the code to report an out-of-range subscript. Optimizing compilers often contain techniques that improve range-checking code.

Checks can be combined. They can be moved out of loops. They can be proved redundant. Taken together, such optimizations can radically reduce the overhead of range checking.

SECTION REVIEW

Programming language implementations store arrays in a variety of formats. The primary ones are contiguous arrays in either row-major or column-major order and disjoint arrays using indirection vectors. Each format has a distinct formula for computing the address of a given element. The address polynomials for contiguous arrays can be optimized with simple algebra to reduce their evaluation costs.

Parameters passed as arrays require cooperation between the caller and the callee. The caller must create a dope vector to hold the information that the callee requires. The caller and callee must agree on the dope vector format.

Review Questions

1. For a two-dimensional array `A` stored in column-major order, write down the address polynomial for the reference `A[i,j]`. Assume that `A` is declared with dimensions $(l_1 : h_1)$ and $(l_2 : h_2)$ and that elements of `A` occupy w bytes.
2. Given an array of integers with dimensions `A[0:99,0:89,0:109]`, how many words of memory are used to represent `A` as a compact row-major order array? How many words are needed to represent `A` using indirection vectors? Assume that both pointers and integers require one word each.

7.6 CHARACTER STRINGS

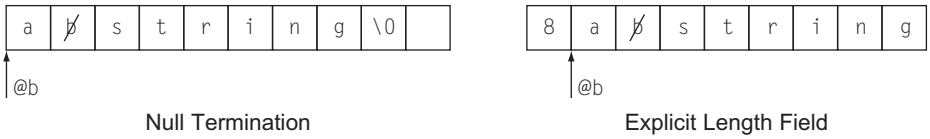
The operations that programming languages provide for character data are different from those provided for numerical data. The level of programming-language support for character strings ranges from C's level of support, where most manipulation takes the form of calls to library routines, to PL/I's level of support, where the language provides first-class mechanisms to assign individual characters, specify arbitrary substrings, and concatenate strings to form new strings. To illustrate the issues that arise in string implementation, this section discusses string assignment, string concatenation, and the string-length computation.

String operations can be costly. Older CISC architectures, such as the IBM S/370 and the DEC VAX, provide extensive support for string manipulation. Modern RISC machines rely more heavily on the compiler to code these

complex operations using a set of simpler operations. The basic operation, copying bytes from one location to another, arises in many different contexts.

7.6.1 String Representations

The compiler writer must choose a representation for strings; the details of that representation have a strong impact on the cost of string operations. To see this point, consider two common representations of a string `b`. The one on the left is traditional in C implementations. It uses a simple vector of characters, with a designated character ('\0') serving as a terminator. The glyph ␣ represents a blank. The representation on the right stores the length of the string (8) alongside its contents. Many language implementations have used this approach.



If the length field takes more space than the null terminator, then storing the length will marginally increase the size of the string in memory. (Our examples assume the length is 4 bytes; in practice, it might be smaller.) However, storing the length simplifies several operations on strings. If a language allows varying-length strings to be stored inside a string allocated with some fixed length, the implementor might also store the allocated length with the string. The compiler can use the allocated length for runtime bounds checking on assignment and concatenation.

7.6.2 String Assignment

String assignment is conceptually simple. In C, an assignment from the third character of `b` to the second character of `a` can be written as `a[1]=b[2];`. On a machine with character-sized memory operations (`cload` and `cstore`), this translates into the simple code shown in the margin. (Recall that the first character in `a` is `a[0]` because C uses zero as the lower bound of all arrays.)

```
loadI    @b     ⇒ r@b
cloadAI  r@b,2  ⇒ r2
loadI    @a     ⇒ r@a
cstoreAI r2     ⇒ r@a,1
```

If, however, the underlying hardware does not support character-oriented memory operations, the compiler must generate more complex code. Assuming that both `a` and `b` begin on word boundaries, that a character occupies 1 byte, and that a word is 4 bytes, the compiler might emit the following code:

```
loadI  0x0000FF00  ⇒ rC2   // mask for 2nd char
loadI  0xFF00FFFF  ⇒ rC124 // mask for chars 1, 2, & 4
```

```
loadI    @b        ⇒ r@b  // address of b
load     r@b       ⇒ r1   // get 1st word of b

and      r1, rC2   ⇒ r2   // mask away others
lshiftI  r2, 8     ⇒ r3   // move it over 1 byte

loadI    @a        ⇒ r@a  // address of a
load     r@a       ⇒ r4   // get 1st word of a

and      r4, rC124 ⇒ r5   // mask away 2nd char
or       r3, r5    ⇒ r6   // put in new 2nd char
store    r6        ⇒ r@a  // put it back in a
```

This code loads the word that contains `b[2]`, extracts the character, shifts it into position, masks it into the proper position in the word that contains `a[1]`, and stores the result back into place. In practice, the masks that the code loads into r_{C2} and r_{C124} would likely be stored in statically initialized storage or computed. The added complexity of this code sequence may explain why character-oriented load and store operations are common.

The code is similar for longer strings. PL/I has a string assignment operator. The programmer can write a statement such as `a = b;` where `a` and `b` have been declared as character strings. Assume that the compiler uses the explicit length representation. The following simple loop will move the characters on a machine with byte-oriented `cload` and `cstore` operations:

```
                 loadI     @b       ⇒ r@b
                 loadAI    r@b, -4  ⇒ r1        // get b's length
                 loadI     @a       ⇒ r@a
                 loadAI    r@a, -4  ⇒ r2        // get a's length
                 cmp_LT    r2, r1   ⇒ r3        // will b fit in a?
                 cbr       r3       → Lsov, L1  // raise overflow
            L1:  loadI     0        ⇒ r4        // counter
a = b;           cmp_LT    r4, r1   ⇒ r5        // more to copy?
                 cbr       r5       → L2, L3
            L2:  cloadAO   r@b, r4  ⇒ r6        // get char from b
                 cstoreAO  r6       ⇒ r@a, r4   // put it in a
                 addI      r4, 1    ⇒ r4        // increment offset
                 cmp_LT    r4, r1   ⇒ r7        // more to copy?
                 cbr       r7       → L2, L3
            L3:  storeAI   r1       ⇒ r@a, -4   // set length
```

Notice that this code tests the lengths of `a` and `b` to avoid overrunning `a`. (With an explicit length representation, the overhead is small.) The label L_{sov} represents a runtime error handler for string-overflow conditions.

In C, which uses null termination for strings, the same assignment would be written as a character-copying loop.

```
t1 = a;
t2 = b;
do {
  *t1++ = *t2++;
} while (*t2 != '\0')
```

```
     loadI   @b      ⇒ r@b    // get pointers
     loadI   @a      ⇒ r@a
     loadI   NULL    ⇒ r1     // terminator
     cload   r@b     ⇒ r2     // get next char
L1:  cstore  r2      ⇒ r@a    // store it
     addI    r@b, 1  ⇒ r@b    // bump pointers
     addI    r@a, 1  ⇒ r@a
     cload   r@b     ⇒ r2     // get next char
     cmp_NE  r1, r2  ⇒ r4
     cbr     r4      → L1, L2
L2:  nop                      // next statement
```

If the target machine supports autoincrement on `load` and `store` operations, the two `add`s in the loop can be performed in the `cload` and `cstore` operations, which reduces the loop to four operations. (Recall that C was originally implemented on the DEC PDP/11, which supported auto-postincrement.) Without autoincrement, the compiler would generate better code by using `cloadAO` and `cstoreAO` with a common offset. That strategy would only use one `add` operation inside the loop.

To achieve efficient execution for long word-aligned strings, the compiler can generate code that uses whole-word loads and stores, followed by a character-oriented loop to handle any leftover characters at the end of the string.

If the processor lacks character-oriented memory operations, the code is more complex. The compiler could replace the load and store in the loop body with a generalization of the scheme for masking and shifting single characters shown in the single character assignment. The result is a functional, but ugly, loop that requires many more instructions to copy `b` into `a`.

The advantages of the character-oriented loops are simplicity and generality. The character-oriented loop handles the unusual but complex cases, such as overlapping substrings and strings with different alignments. The disadvantage of the character-oriented loop is its inefficiency relative to a loop that moves larger blocks of memory on each iteration. In practice, the compiler might well call a carefully optimized library routine to implement the nontrivial cases.

7.6.3 String Concatenation

Concatenation is simply a shorthand for a sequence of one or more assignments. It comes in two basic forms: appending string `b` to string `a`, and creating a new string that contains `a` followed immediately by `b`.

The former case is a length computation followed by an assignment. The compiler emits code to determine the length of `a`. Space permitting, it then performs an assignment of `b` to the space that immediately follows the contents of `a`. (If sufficient space is not available, the code raises an error at runtime.) The latter case requires copying each character in `a` and each character in `b`. The compiler treats the concatenation as a pair of assignments and generates code for the assignments.

In either case, the compiler should ensure that enough space is allocated to hold the result. In practice, either the compiler or the runtime system must know the allocated length of each string. If the compiler knows those lengths, it can perform the check during code generation and avoid the runtime check. In cases where the compiler cannot know the lengths of `a` and `b`, it must generate code to compute the lengths at runtime and to perform the appropriate test and branch.

7.6.4 String Length

Programs that manipulate strings often need to compute a character string's length. In C programs, the function `strlen` in the standard library takes a string as its argument and returns the string's length, expressed as an integer. In PL/I, the built-in function `length` performs the same function. The two string representations described previously lead to radically different costs for the length computation.

1. *Null Terminated String* The length computation must start at the beginning of the string and examine each character, in order, until it reaches the null character. The code is similar to the C character-copying loop. It requires time proportional to the length of the string.
2. *Explicit Length Field* The length computation is a memory reference. In ILOC, this becomes a `loadI` of the string's starting address into a register, followed by a `loadAI` to obtain the length. The cost is constant and small.

The tradeoff between these representations is simple. Null termination saves a small amount of space, but requires more code and more time for the length computation. An explicit length field costs one more word per string, but makes the length computation take constant time.

A classic example of a string optimization problem is finding the length that would result from the concatenation of two strings, `a` and `b`. In a language with string operators, this might be written as `length(a+b)`, where + signifies concatenation. This expression has two obvious implementations: construct the concatenated string and compute its length (`strlen(strcat(a,b))` in C),

and sum the lengths of a and b (strlen(a)+strlen(b) in C). The latter solution, of course, is desired. With an explicit length field, the operation can be optimized to use two loads and an add.

SECTION REVIEW

In principle, string operations are similar to operations on vectors. The details of string representation and the complications introduced by issues of alignment and a desire for efficiency can complicate the code that the compiler must generate. Simple loops that copy one character at a time are easy to generate, to understand, and to prove correct. More complex loops that move multiple characters per iteration can be more efficient; the cost of that efficiency is additional code to handle the end cases. Many compilers simply fall back on a system supplied string-copy routine, such as the Linux strcpy or memmove routines, for the complex cases.

Review Questions

1. Write the ILOC code for the string assignment a ← b using word-length loads and stores. (Use character-length loads and stores in a post loop to clean up the end cases.) Assume that a and b are word aligned and nonoverlapping.
2. How does your code change if a and b are character aligned rather than word aligned? What complications would overlapping strings introduce?

7.7 STRUCTURE REFERENCES

Most programming languages provide a mechanism to aggregate data together into a structure. The C structure is typical; it aggregates individually named elements, often of different types. A list implementation, in C, might, for example, use the following structure to create lists of integers:

```
struct node {
  int value;
  struct node *next;
};

struct node NILNode = {0, (struct node*) 0};
struct node *NIL = &NILNode;
```

Each `node` contains a single integer and a pointer to another node. The final declarations creates a node, `NILNode`, and a pointer, `NIL`. They initialize `NILNode` with value zero and an illegal next pointer, and set `NIL` to point at `NILNode`. (Programs often use a designated `NIL` pointer to denote the end of a list.) The introduction of structures and pointers creates two distinct problems for the compiler: *anonymous values* and *structure layout*.

7.7.1 Understanding Structure Layouts

When the compiler emits code for structure references, it needs to know both the starting address of the structure instance and the offset and length of each structure element. To maintain these facts, the compiler can build a separate table of structure layouts. This compile-time table must include the textual name for each structure element, its offset within the structure, and its source-language data type. For the list example on page 374, the compiler might build the tables shown in Figure 7.12. Entries in the element table use fully qualified names to avoid conflicts due to reuse of a name in several distinct structures.

With this information, the compiler can easily generate code for structure references. Returning to the list example, the compiler might translate the reference `p1->next`, for a pointer to node `p1`, into the following ILOC code:

```
loadI  4          ⇒ r1  // offset of next
loadAO rp1, r1    ⇒ r2  // value of p1->next
```

Structure Layout Table

Name	Length	1st Element
node	8	
...	...	

Structure Element Table

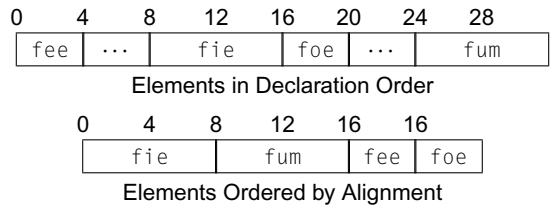
Name	Length	Offset	Type	Next
node.value	4	0	int	
node.next	4	4	struct node *	
...	...	...	...	...

■ **FIGURE 7.12** Structure Tables for the List Example.

Here, the compiler finds the offset of `next` by following the table from the `node` entry in the structure table to the chain of entries for `node` in the element table. Walking that chain, it finds the entry for `node.next` and its offset, 4.

In laying out a structure and assigning offsets to its elements, the compiler must obey the alignment rules of the target architecture. This may force it to leave unused space in the structure. The compiler confronts this problem when it lays out the structure declared on the left:

```
struct example {
  int fee;
  double fie;
  int foe;
  double fum;
};
```



The top-right drawing shows the structure layout if the compiler is constrained to place the elements in declaration order. Because `fie` and `fum` must be doubleword aligned, the compiler must insert padding after `fee` and `foe`. If the compiler could order the elements in memory arbitrarily, it could use the layout shown on the bottom left, which needs no padding. This is a language-design issue: the language definition specifies whether or not the layout of a structure is exposed to the user.

7.7.2 Arrays of Structures

Many programming languages allow the user to declare an array of structures. If the user is allowed to take the address of a structure-valued element of an array, then the compiler must lay out the data in memory as multiple copies of the structure layout. If the programmer cannot take the address of a structure-valued element of an array, the compiler might lay out the structure as if it were a structure composed of elements that are, themselves, arrays. Depending on how the surrounding code accesses the data, these two strategies may have strikingly different performance on a system with cache memory.

To address an array of structures laid out as multiple copies of the structure, the compiler uses the array-address polynomials described in Section 7.5. The overall length of the structure, including any needed padding, becomes the element size w in the address polynomial. The polynomial generates the address of the start of the structure instance. To obtain the value of a specific element, the element's offset is added to the instance's address.

If the compiler has laid out the structure with elements that are arrays, it must compute the starting location of the element array using the offset-table information and the array dimension. This address can then be used as the starting point for an address calculation using the appropriate array-address polynomial.

7.7.3 Unions and Runtime Tags

Many languages allow the programmer to create a structure with multiple, data-dependent interpretations. In C, the union construct has this effect. Pascal achieved the same effect with its variant records.

Unions and variants present one additional complication. To emit code for a reference to an element of a union, the compiler must resolve the reference to a specific offset. Because a union is built from multiple structure definitions, the possibility exists that element names are not unique. The compiler must resolve each reference to a unique offset and type in the runtime object.

This problem has a linguistic solution. The programming language can force the programmer to make the reference unambiguous. Consider the C declarations shown in Figure 7.13. Panel a shows declarations for two kinds of node, one that holds an integer value and another that holds a floating-point value.

The code in panel b declares a union named `one` that is either an `n1` or an `n2`. To reference an integer `value`, the programmer specifies `u1.inode.value`. To reference a floating-point `value`, the programmer specifies `u1.fnode.value`. The fully qualified name resolves any ambiguity.

```
struct n1 {
  int kind;
  int value;
};

struct n2 {
  int kind;
  float value;
};
```

(a) Basic Structures

```
union one {
  struct n1 inode;
  struct n2 fnode;
} u1;
```

(b) Union of Structures

```
union two {
  struct {
    int kind;
    int value;
  } inode;
  struct {
    int kind;
    float value;
  } fnode;
} u2;
```

(c) Union of Implicit Structures

■ **FIGURE 7.13** Union Declarations in C.

The code in panel c declares a union named `two` that has the same properties as `one`. The declaration of `two` explicitly declares its internal structure. The linguistic mechanism for disambiguating a reference to value, however, is the same—the programmer specifies a fully qualified name.

As an alternative, some systems have relied on runtime discrimination. Here, each variant in the union has a field that distinguishes it from all other variants—a "tag." (For example, the declaration of `two`, might initialize `kind` to one for `inode` and to two for `fnode`.) The compiler can then emit code to check the value of the tag field and ensure that each object is handled correctly. In essence, it emits a case statement based on the tag's value. The language may require that the programmer define the tag field and its values; alternatively, the compiler could generate and insert tags automatically. In this latter case, the compiler has a strong motivation to perform type checking and remove as many checks as possible.

7.7.4 Pointers and Anonymous Values

A c program creates an instance of a structure in one of two ways. It can declare a structure instance, as with `NilNode` in the earlier example. Alternatively, the code can explicitly allocate a structure instance. For a variable `fee` declared as a pointer to node, the allocation would look like:

```
fee = (struct node *) malloc(sizeof(node));
```

The only access to this new `node` is through the pointer `fee`. Thus, we think of it as an anonymous value, since it has no permanent name.

Because the only name for an anonymous value is a pointer, the compiler cannot easily determine if two pointer references specify the same memory location. Consider the code fragment

```
1  p1 = (node *) malloc(sizeof(node));
2  p2 = (node *) malloc(sizeof(node));
3  if (...)
4     then p3 = p1;
5     else p3 = p2;
6  p1->value = ...;
7  p3->value = ...;
8  ...   = p1->value;
```

The first two lines create anonymous `node`s. Line 6 writes through `p1` while line 7 writes through `p3`. Because of the `if-then-else`, `p3` can refer to either the node allocated in line 1 or in line 2. Finally, line 8 references `p1->value`.

The use of pointers limits the compiler's ability to keep values in registers. Consider the sequence of assignments in lines 6 through 8. Line 8 reuses either the value assigned in line 6 or the value assigned in line 7. As a matter of efficiency, the compiler should avoid storing that value to memory and reloading it. However, the compiler cannot easily determine which value line 8 uses. The answer to that question depends on the value of the conditional expression in line 3.

While it may be possible to know the value of the conditional expression in certain specific instances (for example, `1 > 2`), it is undecidable in the general case. Unless the compiler knows the value of the conditional expression, it must emit conservative code for the three assignments. It must load the value used in line 8 from memory, even though it recently had the value in a register.

The uncertainty introduced by pointers prevents the compiler from keeping values used in pointer-based references in registers. Anonymous objects further complicate the problem because they introduce an unbounded set of objects to track. As a result, statements that involve pointer-based references are often less efficient than the corresponding computations on unambiguous local values.

A similar effect occurs for code that makes intensive use of arrays. Unless the compiler performs an in-depth analysis of the array subscripts, it may not be able to determine whether two array references overlap. When the compiler cannot distinguish between two references, such as `a[i,j,k]` and `a[i,j,l]`, it must treat both references conservatively. The problem of disambiguating array references, while challenging, is easier than the problem of disambiguating pointer references.

Analysis to disambiguate pointer references and array references is a major source of potential improvement in program performance. For pointer-intensive programs, the compiler may perform an interprocedural data-flow analysis aimed at discovering, for each pointer, the set of objects to which it can point. For array-intensive programs, the compiler may use data-dependence analysis to understand the patterns of array references.

Data-dependence analysis is beyond the scope of this book. See [352, 20, 270].

SECTION REVIEW

To implement structures and arrays of structures, the compiler must establish a layout for each structure and must have a formula to calculate the offset of any structure element. In a language where the declarations dictate the relative position of data elements, structure layout simply requires the compiler to calculate offsets. If the language allows the compiler to determine the relative position of the data elements, then the layout problem is similar to data-area layout (see Section 7.2.2). The address computation for a structure element is a simple application of the schemes used for scalar variables (e.g. base + offset) and for array elements.

Two features related to structures introduce complications. If the language permits unions or variant structures, then input code must specify the desired element in an unambiguous way. The typical solution to this problem is the use of fully qualified names for structure elements in a union. The second issue arises from runtime allocation of structures. The use of pointers to hold addresses of dynamically allocated objects introduces ambiguities that complicate the issue of which values can be kept in registers.

Review Questions

1. When the compiler lays out a structure, it must ensure that each element of the structure is aligned on the appropriate boundary. The compiler may need to insert padding (blank space) between elements to meet alignment restrictions. Write a set of "rules of thumb" that a programmer could use to reduce the likelihood of compiler-inserted padding.
2. If the compiler has the freedom to rearrange structures and arrays, it can sometimes improve performance. What programming language features inhibit the compiler's ability to perform such rearrangement?

7.8 CONTROL-FLOW CONSTRUCTS

A basic block is just a maximal-length sequence of straight-line, unpredicated code. Any statement that does not affect control flow can appear inside a block. Any control-flow transfer ends the block, as does a labelled statement since it can be the target of a branch. As the compiler generates code, it can build up basic blocks by simply aggregating consecutive, unlabeled, non-control-flow operations. (We assume that a labelled statement is not labelled gratuitously, that is, every labelled statement is the target of

some branch.) The representation of a basic block need not be complex. For example, if the compiler has an assembly-like representation held in a simple linear array, then a block can be described by a pair, ⟨*first,last*⟩, that holds the indices of the instruction that begins the block and the instruction that ends the block. (If the block indices are stored in ascending numerical order, an array of *firsts* will suffice.)

To tie a set of blocks together so that they form a procedure, the compiler must insert code that implements the control-flow operations of the source program. To capture the relationships among blocks, many compilers build a control-flow graph (CFG, see Sections 5.2.2 and 8.6.1) and use it for analysis, optimization, and code generation. In the CFG, nodes represent basic blocks and edges represent possible transfers of control between blocks. Typically, the CFG is a derivative representation that contains references to a more detailed representation of each block.

The code to implement control-flow constructs resides in the basic blocks—at or near the end of each block. (In ILOC, there is no fall-through case on a branch, so every block ends with a branch or a jump. If the IR models delay slots, then the control-flow operation may not be the last operation in the block.) While many different syntactic conventions have been used to express control flow, the number of underlying concepts is small. This section examines many of the control-flow constructs found in modern programming languages.

7.8.1 Conditional Execution

Most programming languages provide some version of an `if-then-else` construct. Given the source text

```
if expr
   then statement1
   else statement2
statement3
```

the compiler must generate code that evaluates *expr* and branches to $statement_1$ or $statement_2$, based on the value of *expr*. The ILOC code that implements the two statements must end with a jump to $statement_3$. As we saw in Section 7.4, the compiler has many options for implementing `if-then-else` constructs.

The discussion in Section 7.4 focused on evaluating the controlling expression. It showed how the underlying instruction set influenced the strategies for handling both the controlling expression and, in some cases, the controlled statements.

Programmers can place arbitrarily large code fragments inside the `then` and `else` parts. The size of these code fragments has an impact on the compiler's strategy for implementing the `if-then-else` construct. With trivial `then` and `else` parts, as shown in Figure 7.9, the primary consideration for the compiler is matching the expression evaluation to the underlying hardware. As the `then` and `else` parts grow, the importance of efficient execution inside the `then` and `else` parts begins to outweigh the cost of executing the controlling expression.

For example, on a machine that supports predicated execution, using predicates for large blocks in the `then` and `else` parts can waste execution cycles. Since the processor must issue each predicated instruction to one of its functional units, each operation with a false predicate has an opportunity cost—it ties up an issue slot. With large blocks of code under both the `then` and `else` parts, the cost of unexecuted instructions may outweigh the overhead of using a conditional branch.

Figure 7.14 illustrates this tradeoff. It assumes that both the `then` and `else` parts contain 10 independent ILOC operations and that the target machine can issue two operations per cycle.

Figure 7.14a shows code that might be generated using predication; it assumes that the value of the controlling expression is in r_1. The code issues two instructions per cycle. One of them executes in each cycle. All of the `then` part's operations are issued to Unit 1, while the `then` part's operations are issued to Unit 2. The code avoids all branching. If each operation

Unit 1	Unit 2
comparison $\Rightarrow r_1$	
(r_1) op_1	$(\neg r_1)$ op_{11}
(r_1) op_2	$(\neg r_1)$ op_{12}
(r_1) op_3	$(\neg r_1)$ op_{13}
(r_1) op_4	$(\neg r_1)$ op_{14}
(r_1) op_5	$(\neg r_1)$ op_{15}
(r_1) op_6	$(\neg r_1)$ op_{16}
(r_1) op_7	$(\neg r_1)$ op_{17}
(r_1) op_8	$(\neg r_1)$ op_{18}
(r_1) op_9	$(\neg r_1)$ op_{19}
(r_1) op_{10}	$(\neg r_1)$ op_{20}

(a) Using Predicates

Unit 1	Unit 2
compare & branch	
L_1: op_1	op_2
op_3	op_4
op_5	op_6
op_7	op_8
op_9	op_{10}
jumpI	$\rightarrow L_3$
L_2: op_{11}	op_{12}
op_{13}	op_{14}
op_{15}	op_{16}
op_{17}	op_{18}
op_{19}	op_{20}
jumpI	$\rightarrow L_3$
L_3: nop	

(b) Using Branches

■ **FIGURE 7.14** Predication versus Branching.

BRANCH PREDICTION BY USERS

One urban compiler legend concerns branch prediction. FORTRAN has an arithmetic `if` statement that takes one of three branches, based on whether the controlling expression evaluates to a negative number, to zero, or to a positive number. One early compiler allowed the user to supply a weight for each label that reflected the relative probability of taking that branch. The compiler then used the weights to order the branches in a way that minimized total expected delay from branching.

After the compiler had been in the field for a year, the story goes, a maintainer discovered that the branch weights were being used in the reverse order, maximizing the expected delay. No one had complained. The story is usually told as a fable about the value of programmers' opinions about the behavior of code they have written. (Of course, no one reported the improvement, if any, from using the branch weights in the correct order.)

takes a single cycle, it takes 10 cycles to execute the controlled statements, independent of which branch is taken.

Figure 7.14b shows code that might be generated using branches; it assumes that control flows to L_1 for the `then` part or to L_2 for the `else` part. Because the instructions are independent, the code issues two instructions per cycle. Following the `then` path takes five cycles to execute the operations for the taken path, plus the cost of the terminal jump. The cost for the `else` part is identical.

The predicated version avoids the initial branch required in the unpredicated code (to either L_1 or L_2 in the figure), as well as the terminal jumps (to L_3). The branching version incurs the overhead of a branch and a jump, but may execute faster. Each path contains a conditional branch, five cycles of operations, and the terminal jump. (Some of the operations may be used to fill delay slots on jumps.) The difference lies in the effective issue rate—the branching version issues roughly half the instructions of the predicated version. As the code fragments in the `then` and `else` parts grow larger, this difference becomes larger.

Choosing between branching and predication to implement an `if-then-else` requires some care. Several issues should be considered, as follows:

1. *Expected frequency of execution* If one side of the conditional executes significantly more often, techniques that speed execution of that path may produce faster code. This bias may take the form of predicting a branch, of executing some instructions speculatively, or of reordering the logic.

2. *Uneven amounts of code* If one path through the construct contains many more instructions than the other, this may weigh against predication or for a combination of predication and branching.
3. *Control flow inside the construct* If either path contains nontrivial control flow, such as an `if-then-else`, loop, case statement, or call, then predication may be a poor choice. In particular, nested `if` constructs create complex predicates and lower the fraction of issued operations that are useful.

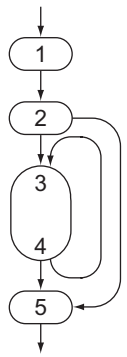
To make the best decision, the compiler must consider all these factors, as well as the surrounding context. These factors may be difficult to assess early in compilation; for example, optimization may change them in significant ways.

7.8.2 Loops and Iteration

Most programming languages include loop constructs to perform iteration. The first FORTRAN compiler introduced the `do` loop to perform iteration. Today, loops are found in many forms. For the most part, they have a similar structure.

Consider the C `for` loop as an example. Figure 7.15 shows how the compiler might lay out the code. The `for` loop has three controlling expressions: e_1, which provides for initialization; e_2, which evaluates to a boolean and governs execution of the loop; and e_3, which executes at the end of each iteration and, potentially, updates the values used in e_2. We will use this figure as the basic schema to explain the implementation of several kinds of loops.

```
For (e1; e2; e3) {
    loop body
}
```



Step	Purpose
1	Evaluate e_1
2	If $(\neg e_2)$ Then goto 5
3	*Loop Body*
4	Evaluate e_3 If (e_2) Then goto 3
5	*Code After Loop*

(a) Example Code for Loop (b) Schema for Implementing Loop

■ **FIGURE 7.15** General Schema for Layout of a `for` Loop.

If the loop body consists of a single basic block—that is, it contains no other control flow—then the loop that results from this schema has an initial branch plus one branch per iteration. The compiler might hide the latency of this branch in one of two ways. If the architecture allows the compiler to predict whether or not the branch is taken, the compiler should predict the branch in step 4 as being taken (to start the next iteration). If the architecture allows the compiler to move instructions into the delay slot(s) of the branch, the compiler should attempt to fill the delay slot(s) with instruction(s) from the loop body.

For Loops

To map a `for` loop into code, the compiler follows the general schema from Figure 7.15. To make this concrete, consider the following example. Steps 1 and 2 produce a single basic block, as shown in the following code:

```
for (i=1; i<=100; i++) {
   loop body
}
next statement
```

```
        loadI   1       ⇒ ri      // Step 1
        loadI   100     ⇒ r1      // Step 2
        cmp_GT  ri , r1 ⇒ r2
        cbr     r2      → L2 , L1
L1:     loop body                 // Step 3
        addI    ri , 1  ⇒ ri      // Step 4
        cmp_LE  ri , r1 ⇒ r3
        cbr     r3      → L1 , L2
L2:     next statement            // Step 5
```

The code produced in steps 1, 2, and 4 is straightforward. If the loop body (step 3) either consists of a single basic block or it ends with a single basic block, then the compiler can optimize the update and test produced in step 4 with the loop body. This may lead to improvements in the code—for example, the instruction scheduler might use operations from the end of step 3 to fill delay slots in the branch from step 4.

The compiler can also shape the loop so that it has only one copy of the test—the one in step 2. In this form, step 4 evaluates e_3 and then jumps to step 2. The compiler would replace `cmp_LE`, `cbr` sequence at the end of the loop with a `jumpI`. This form of the loop is one operation smaller than the two-test form. However, it creates a two-block loop for even the simplest loops, and it lengthens the path through the loop by at least one operation. When code size is a serious consideration, consistent use of this more compact loop form might be worthwhile. As long as the loop-ending jump is an immediate jump, the hardware can take steps to minimize any disruption that it might cause.

The canonical loop shape from Figure 7.15 also sets the stage for later optimization. For example, if e_1 and e_2 contain only known constants, as in

the example, the compiler can fold the value from step 1 into the test in step 2 and either eliminate the compare and branch (if control enters the loop) or eliminate the loop body (if control never enters the loop). In the single-test loop, the compiler cannot do this. Instead, the compiler finds two paths leading to the test—one from step 1 and one from step 4. The value used in the test, r_i, has a varying value along the edge from step 4, so the test's outcome is not predictable.

FORTRAN's do Loop

In FORTRAN, the iterative loop is a `do` loop. It resembles the C `for` loop, but has a more restricted form.

```
   j = 1
   do 10 i = 1, 100
      loop body
      j = j + 2
10    continue
   next statement
```

```
        loadI  1      ⇒ rj     // j ← 1
        loadI  1      ⇒ ri     // Step 1
        loadI  100    ⇒ r1     // Step 2
        cmp_GT ri, r1 ⇒ r2
        cbr    r2     → L2, L1
L1:     loop body              // Step 3
        addI   rj, 2  ⇒ rj     // j ← j+2
        addI   ri, 1  ⇒ ri     // Step 4
        cmp_LE ri, r1 ⇒ r3
        cbr    r3     → L1, L2
L2:     next statement         // Step 5
```

The comments map portions of the ILOC code back to the schema in Figure 7.15.

The definition of FORTRAN, like that of many languages, has some interesting quirks. One such peculiarity relates to `do` loops and their index variables. The number of iterations of a loop is fixed before execution enters the loop. If the program changes the index variable's value, that change does not affect the number of iterations that execute. To ensure the correct behavior, the compiler may need to generate a hidden induction variable, called a *shadow index variable*, to control the iteration.

While Loops

A `while` loop can also be implemented with the loop schema in Figure 7.15. Unlike the C `for` loop or the FORTRAN `do` loop, a `while` loop has no initialization. Thus, the code is even more compact.

```
while (x < y) {
   loop body
}
next statement
```

```
        cmp_LT rx, ry ⇒ r1    // Step 2
        cbr    r1     → L1, L2
L1:     loop body             // Step 3
        cmp_LT rx, ry ⇒ r2    // Step 4
        cbr    r2     → L1, L2
L2:     next statement        // Step 5
```

Replicating the test in step 4 creates the possibility of a loop with a single basic block. The same benefits that accrue to a `for` loop from this structure also occur for a `while` loop.

Until Loops

An `until` loop iterates as long as the controlling expression is false. It checks the controlling expression after each iteration. Thus, it always enters the loop and performs at least one iteration. This produces a particularly simple loop structure, since it avoids steps 1 and 2 in the schema:

```
{
   loop body
} until (x < y)
next statement
```

```
L1: loop body                  // Step 3
    cmp_LT rx, ry ⇒ r2         // Step 4
    cbr    r2     → L2, L1
L2: next statement             // Step 5
```

C does not have an `until` loop. Its `do` construct is similar to an `until` loop, except that the sense of the condition is reversed. It iterates as long as the condition evaluates to true, where the `until` iterates as long as the condition is false.

Expressing Iteration as Tail Recursion

In Lisp-like languages, iteration is often implemented (by programmers) using a stylized form of recursion. If the last action executed by a function is a call, that call is known as a *tail call*. For example, to find the last element of a list in Scheme, the programmer might write the following simple function:

Tail call
A procedure call that occurs as the last action in some procedure is termed a tail call. A self-recursive tail call is termed a *tail recursion*.

```
(define (last alon)
   (cond
      ((empty? alon) empty)
      ((empty? (cdr alon)) (car alon))
      (else (last (cdr alon)))))
```

Compilers often subject tail calls to special treatment, because the compiler can generate particularly efficient call for them (see Section 10.4.1). Tail recursion can be used to achieve the same effects as iteration, as in the following Scheme code:

```
(define (count alon ct)
   (cond
      ((empty? alon) ct)
      (else (count (cdr alon) (+ ct 1)))))

(define (len alon)
   (count alon 0))
```

Invoking `len` on a list returns the list's length. `len` relies on `count`, which implements a simple counter using tail calls.

Break Statements

Several languages implement variations on a `break` or `exit` statement. The `break` statement is a structured way to exit a control-flow construct. In a loop, `break` transfers control to the first statement following the loop. For nested loops, a `break` typically exits the innermost loop. Some languages, such as Ada and Java, allow an optional label on a `break` statement. This causes the `break` statement to exit from the enclosing construct specified by that label. In a nested loop, a labelled `break` allows the program to exit several loops at once. C also uses `break` in its `switch` statement, to transfer control to the statement that follows the `switch` statement.

These actions have simple implementations. Each loop and each case statement should end with a label for the statement that follows it. A `break` would be implemented as an immediate jump to that label. Some languages include a `skip` or `continue` statement that jumps to the next iteration of a loop. This construct can be implemented as an immediate jump to the code that reevaluates the controlling expression and tests its value. Alternatively, the compiler can simply insert a copy of the evaluation, test, and branch at the point where the `skip` occurs.

7.8.3 Case Statements

Many programming languages include some variant of a case statement. FORTRAN has its computed goto. Algol-W introduced the case statement in its modern form. BCPL and C have a `switch` construct, while PL/I has a generalized construct that maps well onto a nested set of `if-then-else` statements. As the introduction to this chapter hinted, implementing a case statement efficiently is complex.

Consider the implementation of C's `switch` statement. The basic strategy is straightforward: (1) evaluate the controlling expression; (2) branch to the selected case; and (3) execute the code for that case. Steps 1 and 3 are well understood, as they follow from discussions elsewhere in this chapter. In C, the individual cases usually end with a `break` statement that exits the `switch` statement.

The complex part of case-statement implementation lies in choosing an efficient method to locate the designated case. Because the desired case is not known until runtime, the compiler must emit code that will use the value of the controlling expression to locate the corresponding case. No single

```
switch (e1) {
  case 0:  block0;
           break;
  case 1:  block1;
           break;
  case 3:  block3;
           break;
  default: blockd;
           break;
}
```

(a) Switch Statement

```
t1 ← e1
if (t1 = 0)
   then block0
   else if (t1 = 1)
      then block1
      else if (t1 = 2)
         then block2
         else if (t1 = 3)
            then block3
            else blockd
```

(b) Implemented as a Linear Search

■ **FIGURE 7.16** Case Statement Implemented with Linear Search.

method works well for all case statements. Many compilers have provision for several different search schemes and choose between them based on the specific details of the set of cases.

This section examines three strategies: a linear search, a binary search, and a computed address. Each strategy is appropriate under different circumstances.

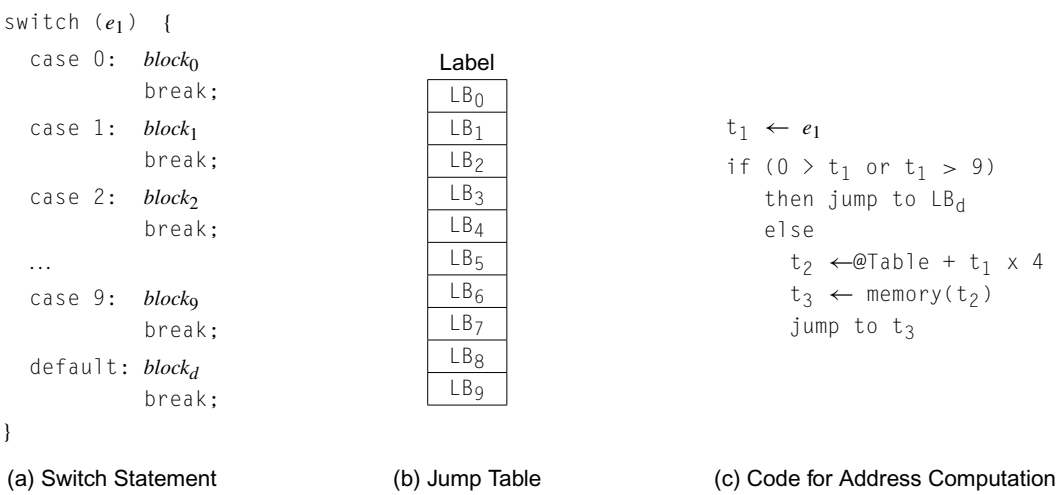
Linear Search

The simplest way to locate the appropriate case is to treat the case statement as the specification for a nested set of `if-then-else` statements. For example, the `switch` statement shown in Figure 7.16a can be translated into the nest of statements shown in Figure 7.16b. This translation preserves the meaning of the `switch` statement, but makes the cost of reaching individual cases dependent on the order in which they are written. With a linear search strategy, the compiler should attempt to order the cases by estimated execution frequency. Still, when the number of cases is small—say three or four—this strategy can be efficient.

Directly Computing the Address

If the case labels form a compact set, the compiler can do better than binary search. Consider the `switch` statement shown in Figure 7.17a. It has case labels from zero to nine, plus a default case. For this code, the compiler can build a compact vector, or *jump table*, that contains the block labels, and find the appropriate label by index into the table. The jump table is shown

Jump table
a vector of labels used to transfer control based on a computed index into the table



■ **FIGURE 7.17** Case Statement Implemented with Direct Address Computation.

in Figure 7.17b, while the code to compute the correct case's label is shown in Figure 7.17c. The search code assumes that the jump table is stored at `@Table` and that each label occupies four bytes.

For a dense label set, this scheme generates compact and efficient code. The cost is small and constant—a brief calculation, a memory reference, and a `jump`. If a few holes exist in the label set, the compiler can fill those slots with the label for the default case. If no default case exists, the appropriate action depends on the language. In C, for example, the code should branch to the first statement after the `switch`, so the compiler can place that label in each hole in the table. If the language treats a missing case as an error, as PL/I did, the compiler can fill holes in the jump table with the label of a block that throws the appropriate runtime error.

Binary Search

As the number of cases rises, the efficiency of linear search becomes a problem. In a similar way, as the label set becomes less dense and less compact, the size of the jump table can become a problem for the direct address computation. The classic solutions that arise in building an efficient search apply in this situation. If the compiler can impose an order on the case labels, it can use binary search to obtain a logarithmic search rather than a linear one.

The idea is simple. The compiler builds a compact ordered table of case labels, along with their corresponding branch labels. It uses binary search to

```
switch (e1)  {
  case 0:   block0
            break;
  case 15:  block15
            break;
  case 23:  block23
            break;
  ...
  case 99:  block99
            break;
  default:  blockd
            break;
}
```

(a) Switch Statement

Value	Label
0	LB_0
15	LB_{15}
23	LB_{23}
37	LB_{37}
41	LB_{41}
50	LB_{50}
68	LB_{68}
72	LB_{72}
83	LB_{83}
99	LB_{99}

(b) Search Table

```
t1 ← e1
down ← 0   // lower bound
up ← 10    // upper bound + 1
while (down + 1 < up)  {
   middle ← (up + down) ÷ 2
   if (Value [middle] ≤ t1)
      then down ← middle
      else up ← middle
}
if (Value [down] = t1
   then jump to Label[down]
   else jump to LBd
```

(c) Code for Binary Search

■ **FIGURE 7.18** Case Statement Implemented with Binary Search.

discover a matching case label, or the absence of a match. Finally, it either branches to the corresponding label or to the `default` case.

Figure 7.18a shows our example case statement, rewritten with a different set of labels. For the figure, we will assume case labels of 0, 15, 23, 37, 41, 50, 68, 72, 83, and 99, as well as a default case. The labels could, of course, cover a much larger range. For such a case statement, the compiler might build a search table such as the one shown in Figure 7.18b, and generate a binary search, as in Figure 7.18c, to locate the desired case. If fall-through behavior is allowed, as in C, the compiler must ensure that the blocks appear in memory in their original order.

The exact form of the search loop might vary. For example, the code in the figure does not short circuit the case when it finds the label early. Empirical testing of several variants written in the target machine's assembly code is needed to find the best choices.

In a binary search or direct address computation, the compiler writer should ensure that the set of potential targets of the jump are visible in the IR, using a construct such as the ILOC `tbl` pseudo-operation (see Appendix A.4.2). Such hints both simplify later analysis and make its results more precise.

SECTION REVIEW

Programming languages include a variety of features to implement control flow. The compiler needs a schema for each control-flow construct in the source languages that it accepts. In some cases, such as a loop, one approach serves for a variety of different constructs. In others, such as a case statement, the compiler should choose an implementation strategy based on the specific properties of the code at hand.

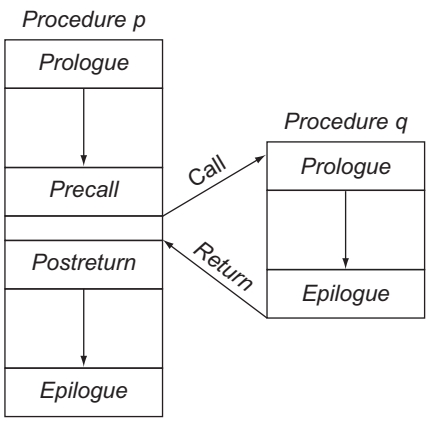
```
    do 10 i = 1, 100
       loop body
       i = i + 2
10     continue
```

Review Questions

1. Write the ILOC code for the FORTRAN loop shown in the margin. Recall that the loop body must execute 100 iterations, even though the loop modifies the value of `i`.
2. Consider the tradeoff between implementing a C switch statement with a direct address computation and with a binary search. At what point should the compiler switch from direct address computation to a binary search? What properties of the actual code should play a role in that determination?

7.9 PROCEDURE CALLS

The implementation of procedure calls is, for the most part, straightforward. As shown in Figure 7.19, a procedure call consists of a precall sequence and a postreturn sequence in the caller, and a prologue and an epilogue in the callee. A single procedure can contain multiple call sites, each with its own precall and postreturn sequences. In most languages, a procedure has one entry point, so it has one prologue sequence and one epilogue sequence. (Some languages allow multiple entry points, each of which has its own prologue sequence.) Many of the details involved in these sequences are described in Section 6.5. This section focuses on issues that affect the compiler's ability to generate efficient, compact, and consistent code for procedure calls.



■ **FIGURE 7.19** A Standard Procedure Linkage.

As a general rule, moving operations from the precall and postreturn sequences into the prologue and epilogue sequences should reduce the overall size of the final code. If the call from p to q shown in Figure 7.19 is the only call to q in the entire program, then moving an operation from the precall sequence in p to the prologue in q (or from the postreturn sequence in p to the epilogue in q) has no impact on code size. If, however, other call sites invoke q and the compiler moves an operation from the caller to the callee (at all the call sites), it should reduce the overall code size by replacing multiple copies of an operation with a single one. As the number of call sites that invoke a given procedure rises, the savings grow. We assume that most procedures are called from several locations; if not, both the programmer and the compiler should consider including the procedure inline at the point of its only invocation.

From the code-shape perspective, procedure calls are similar in Algol-like languages and object-oriented languages. The major difference between them lies in the technique used to name the callee (see Section 6.3.4). In addition, a call in an object-oriented language typically adds an implicit actual parameter, that is, the receiver's object record.

7.9.1 Evaluating Actual Parameters

When it builds the precall sequence, the compiler must emit code to evaluate the actual parameters to the call. The compiler treats each actual parameter as an expression. For a call-by-value parameter, the precall sequence evaluates the expression and stores its value in a location designated for that parameter—either in a register or in the callee's AR. For a call-by-reference parameter, the precall sequence evaluates the parameter to an address and stores the address in a location designated for that parameter. If a call-by-reference parameter has no storage location, then the compiler may need to allocate space to hold the parameter's value so that it has an address to pass to the callee.

If the source language specifies an order of evaluation for the actual parameters, the compiler must, of course, follow that order. Otherwise, it should use a consistent order—either left to right or right to left. The evaluation order matters for parameters that might have side effects. For example, a program that used two routines `push` and `pop` to manipulate a stack would produce different results for the sequence `subtract(pop(),pop())` under left-to-right and right-to-left evaluation.

Procedures typically have several implicit arguments. These include the procedure's ARP, the caller's ARP, the return address, and any information

needed to establish addressability. Object-oriented languages pass the receiver as an implicit parameter. Some of these arguments are passed in registers while others usually reside in memory. Many architectures have an operation like

$$\texttt{jsr label}_1 \Rightarrow \texttt{r}_i$$

that transfers control to $\texttt{label}_1$ and places the address of the operation that follows the `jsr` into $\texttt{r}_i$.

Procedures passed as actual parameters may require special treatment. If *p* calls *q*, passing procedure *r* as an argument, *p* must pass to *q* more information than *r*'s starting address. In particular, if the compiled code uses access links to find nonlocal variables, the callee needs *r*'s lexical level so that a subsequent call to *r* can find the correct access link for *r*'s level. The compiler can construct an ⟨*address,level*⟩ pair and pass it (or its address) in place of the procedure-valued parameter. When the compiler constructs the precall sequence for a procedure-valued parameter, it must insert the extra code to fetch the lexical level and adjust the access link accordingly.

7.9.2 Saving and Restoring Registers

Under any calling convention, one or both of the caller and the callee must preserve register values. Often, linkage conventions use a combination of caller-saves and callee-saves registers. As both the cost of memory operations and the number of registers have risen, the cost of saving and restoring registers at call sites has increased, to the point where it merits careful attention.

In choosing a strategy to save and restore registers, the compiler writer must consider both efficiency and code size. Some processor features impact this choice. Features that spill a portion of the register set can reduce code size. Examples of such features include register windows on the SPARC machines, the multiword load and store operations on the Power architectures, and the high-level call operation on the VAX. Each offers the compiler a compact way to save and restore some portion of the register set.

While larger register sets can increase the number of registers that the code saves and restores, in general, using these additional registers improves the speed of the resulting code. With fewer registers, the compiler would be forced to generate loads and stores throughout the code; with more registers,

many of these spills occur only at a call site. (The larger register set should reduce the total number of spills in the code.) The concentration of saves and restores at call sites presents the compiler with opportunities to handle them in better ways than it might if they were spread across an entire procedure.

- *Using multi-register memory operations* When saving and restoring adjacent registers, the compiler can use a multiregister memory operation. Many ISAs support doubleword and quadword load and store operations. Using these operations can reduce code size; it may also improve execution speed. Generalized multiregister memory operations can have the same effect.
- *Using a library routine* As the number of registers grows, the precall and postreturn sequences both grow. The compiler writer can replace the sequence of individual memory operations with a call to a compiler-supplied save or restore routine. Done across all calls, this strategy can produce a significant savings in code size. Since the save and restore routines are known only to the compiler, they can use minimal call sequence to keep the runtime cost low.

  The save and restore routines can take an argument that specifies which registers must be preserved. It may be worthwhile to generate optimized versions for common cases, such as preserving all the caller-saves or callee-saves registers.
- *Combining responsibilities* To further reduce overhead, the compiler might combine the work for caller-saves and callee-saves registers. In this scheme, the caller passes a value to the callee that specifies which registers it must save. The callee adds the registers it must save to the value and calls the appropriate compiler-provided save routine. The epilogue passes the same value to the restore routine so that it can reload the needed registers. This approach limits the overhead to one call to save registers and one to restore them. It separates responsibility (caller saves versus callee saves) from the cost to call the routine.

The compiler writer must pay close attention to the implications of the various options on code size and runtime speed. The code should use the fastest operations for saves and restores. This requires a close look at the costs of single-register and multiregister operations on the target architecture. Using library routines to perform saves and restores can save space; careful implementation of those library routines may mitigate the added cost of invoking them.

SECTION REVIEW

The code generated for procedure calls is split between the caller and the callee, and between the four pieces of the linkage sequence (prologue, epilogue, precall, and postreturn). The compiler coordinates the code in these multiple locations to implement the linkage convention, as discussed in Chapter 6. Language rules and parameter binding conventions dictate the order of evaluation and the style of evaluation for actual parameters. System-wide conventions determine responsibility for saving and restoring registers.

Compiler writers pay particular attention to the implementation of procedure calls because the opportunities are difficult for general optimization techniques (see Chapters 8 and 10) to discover. The many-to-one nature of the caller-callee relationship complicates analysis and transformation, as does the distributed nature of the cooperating code sequences. Equally important, minor deviations from the defined linkage convention can cause incompatibilities in code compiled with different compilers.

Review Questions

1. When a procedure saves registers, either callee-saves registers in its prologue or caller-saves registers in a precall sequence, where should it save those registers? Are all of the registers saved for some call stored in the same AR?
2. In some situations, the compiler must create a storage location to hold the value of a call-by-reference parameter. What kinds of parameters may not have their own storage locations? What actions might be required in the precall and postcall sequences to handle these actual parameters correctly?

7.10 SUMMARY AND PERSPECTIVE

One of the more subtle tasks that confronts the compiler writer is selecting a pattern of target-machine operations to implement each source-language construct. Multiple implementation strategies are possible for almost any source-language statement. The specific choices made at design time have a strong impact on the code that the compiler generates.

In a compiler that is not intended for production use—a debugging compiler or a student compiler—the compiler writer might select easy to implement translations for each strategy that produce simple, compact code. In

an optimizing compiler, the compiler writer should focus on translations that expose as much information as possible to the later phases of the compiler—low-level optimization, instruction scheduling, and register allocation. These two different perspectives lead to different shapes for loops, to different disciplines for naming temporary variables, and, possibly, to different evaluation orders for expressions.

The classic example of this distinction is the `case` statement. In a debugging compiler, the implementation as a cascaded series of `if-then-else` constructs is fine. In an optimizing compiler, the inefficiency of the myriad tests and branches makes a more complex implementation scheme worthwhile. The effort to improve the `case` statement must be made when the IR is generated; few, if any, optimizers will convert a cascaded series of conditionals into a binary search or a direct jump table.

■ CHAPTER NOTES

The material contained in this chapter falls, roughly, into two categories: generating code for expressions and handling control-flow constructs. Expression evaluation is well explored in the literature. Discussions of how to handle control flow are rarer; much of the material on control flow in this chapter derives from folklore, experience, and careful reading of the output of compilers.

Floyd presented the first multipass algorithm for generating code from expression trees [150]. He points out that both redundancy elimination and algebraic reassociation have the potential to improve the results of his algorithm. Sethi and Ullman [311] proposed a two-pass algorithm that is optimal for a simple machine model; Proebsting and Fischer extended this work to account for small memory latencies [289]. Aho and Johnson [5] introduced dynamic programming to find least-cost implementations.

The predominance of array calculations in scientific programs led to work on array-addressing expressions and to optimizations (like strength reduction, Section 10.7.2) that improve them. The computations described in Section 7.5.3 follow Scarborough and Kolsky [307].

Harrison used string manipulation as a motivating example for the pervasive use of inline substitution and specialization [182]. The example mentioned at the end of Section 7.6.4 comes from that paper.

Mueller and Whalley describe the impact of different loop shapes on performance [271]. Bernstein provides a detailed discussion of the options that arise in generating code for case statements [40]. Calling conventions are best described in processor-specific and operating-system-specific manuals.

Optimization of range checks has a long history. The PL/.8 compiler insisted on checking every reference; optimization lowered the overhead [257]. More recently, Gupta and others have extended these ideas to increase the set of checks that can be moved to compile time [173].

EXERCISES

Section 7.2

1. Memory layout affects the addresses assigned to variables. Assume that character variables have no alignment restriction, short integer variables must be aligned to halfword (2 byte) boundaries, integer variables must be aligned to word (4 byte) boundaries, and long integer variables must be aligned to doubleword (8 byte) boundaries. Consider the following set of declarations:

```
char a;
long int b;
int c;
short int d;
long int e;
char f;
```

Draw a memory map for these variables:

a. Assuming that the compiler cannot reorder the variables

b. Assuming the compiler can reorder the variables to save space

2. As demonstrated in the previous question, the compiler needs an algorithm to lay out memory locations within a data area. Assume that the algorithm receives as input a list of variables, their lengths, and their alignment restrictions, such as

⟨a, 4, 4⟩, ⟨b, 1, 3⟩, ⟨c, 8, 8⟩, ⟨d, 4, 4⟩, ⟨e, 1, 4⟩, ⟨f, 8, 16⟩, ⟨g, 1, 1⟩.

The algorithm should produce, as output, a list of variables and their offsets in the data area. The goal of the algorithm is to minimize unused, or wasted, space.

a. Write down an algorithm to lay out a data area with minimal wasted space.

b. Apply your algorithm to the example list above and two other lists that you design to demonstrate the problems that can arise in storage layout.

c. What is the complexity of your algorithm?

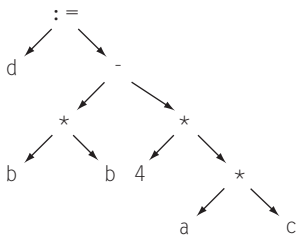
3. For each of the following types of variable, state where in memory the compiler might allocate the space for such a variable. Possible

answers include registers, activation records, static data areas (with different visibilities), and the runtime heap.

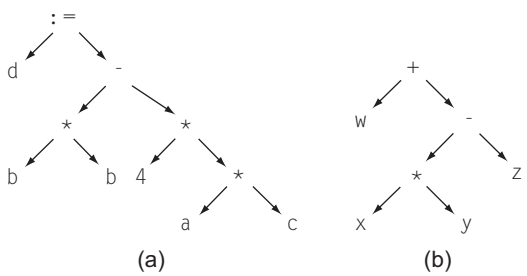
a. A variable local to a procedure
b. A global variable
c. A dynamically allocated global variable
d. A formal parameter
e. A compiler-generated temporary variable

4. Use the treewalk code-generation algorithm from Section 7.3 to generate naive code for the following expression tree. Assume an unlimited set of registers.

Section 7.3



5. Find the minimum number of registers required to evaluate the following trees using the ILOC instruction set. For each nonleaf node, indicate which of its children must be evaluated first in order to achieve this minimum number of registers.



6. Build expression trees for the following two arithmetic expressions, using standard precedence and left-to-right evaluation. Compute the minimum number of registers required to evaluate each of them using the ILOC instruction set.

a. `((a + b) + (c + d)) + ((e + f) + (g + h))`
b. `a + b + c + d + e + f + g + h`

Section 7.4

7. Generate predicated ILOC for the following code sequence. (No branches should appear in the solution.)

```
if (x < y)
   then z = x * 5;
   else z = y * 5;
w = z + 10;
```

8. As mentioned in Section 7.4, short-circuit code for the following expression in C avoids a potential division-by-zero error:

```
a != 0 && b / a > 0.5
```

If the source-language definition does not specify short-circuit evaluation for boolean-valued expressions, can the compiler generate short-circuit code as an optimization for such expressions? What problems might arise?

Section 7.5

9. For a character array `A[10...12,1...3]` stored in row-major order, calculate the address of the reference `A[i,j]`, using at most four arithmetic operations in the generated code.

10. What is a dope vector? Give the contents of the dope vector for the character array in the previous question. Why does the compiler need a dope vector?

11. When implementing a C compiler, it might be advisable to have the compiler perform range checking for array references. Assuming range checks are used and that all array references in a C program have successfully passed them, is it possible for the program to access storage outside the range of an array, for example, accessing `A[-1]` for an array declared with lower bound zero and upper bound `N`?

Section 7.6

12. Consider the following character-copying loop from Section 7.6.2:

```
do {
  *a++ = *b++;
} while (*b!='\0')
```

```
     loadI  @b     ⇒ r@b   // get pointers
     loadI  @a     ⇒ r@a
     loadI  NULL   ⇒ r1    // terminator
L1:  cload  r@b    ⇒ r2    // get next char
     cstore r2     ⇒ r@a   // store it
     addI   r@b,1  ⇒ r@b   // bump pointers
     addI   r@a,1  ⇒ r@a
     cmp_NE r1,r2  ⇒ r4
     cbr    r4     → L1,L2
L2:  nop                   // next stmt
```

Modify the code so that it branches to an error handler at `Lsov` on any attempt to overrun the allocated length of `a`. Assume that the allocated length of `a` is stored as an unsigned four-byte integer at an offset of –8 from the start of `a`.

13. Arbitrary string assignments can generate misaligned cases.

a. Write the ILOC code that you would like your compiler to emit for an arbitrary PL/I-style character assignment, such as

```
fee(i:j) = fie(k:l);
```

where `j-i = l-k`. This statement copies the characters in `fie`, starting at location `k` and running through location `l` into the string `fee`, starting at location `i` and running through location `j`.

Include versions using character-oriented memory operations and versions using word-oriented memory operations. You may assume that `fee` and `fie` do not overlap in memory.

b. The programmer can create character strings that overlap. In PL/I, the programmer might write

```
fee(i:j) = fee(i+1:j+1);
```

or, even more diabolically,

```
fee(i+k:j+k) = fee(i:j);
```

How does this complicate the code that the compiler must generate for the character assignment?

c. Are there optimizations that the compiler could apply to the various character-copying loops that would improve runtime behavior? How would they help?

14. Consider the following type declarations in C: Section 7.7

```
struct S2 {
  int i;
  int f;
};

union U {
   float r;
   struct S2;
};

struct S1 {
   int a;
   double b;
   union U;
   int d;
};
```

Build a structure-element table for `S1`. Include in it all the information that a compiler would need to generate references to elements of a

variable of type S1, including the name, length, offset, and type of each element.

15. Consider the following declarations in C:

```
struct record {
  int StudentId;
  int CourseId;
  int Grade;
} grades[1000];
int g, i;
```

Show the code that a compiler would generate to store the value in variable g as the grade in the i^{th} element of grades, assuming the following:

a. The array grades is stored as an array of structures.

b. The array grades is stored as a structure of arrays.

Section 7.8

16. As a programmer, you are interested in the efficiency of the code that you produce. You recently implemented, by hand, a scanner. The scanner spends most of its time in a single while loop that contains a large case statement.

a. How would the different case statement implementation techniques affect the efficiency of your scanner?

b. How would you change your source code to improve the runtime performance under each of the case statement implementation strategies?

17. Convert the following C tail-recursive function to a loop:

```
List * last(List *l) {
  if (l == NULL)
     return NULL;
  else if (l->next == NULL)
     return l;
  else
     return last(l->next); }
```

Section 7.9

18. Assume that x is an unambiguous, local, integer variable and that x is passed as a call-by-reference actual parameter in the procedure where it is declared. Because it is local and unambiguous, the compiler might try to keep it in a register throughout its lifetime. Because it is

passed as a call-by-reference parameter, it must have a memory address at the point of the call.

a. Where should the compiler store *x*?

b. How should the compiler handle *x* at the call site?

c. How would your answers change if *x* was passed as a call-by-value parameter?

19. The linkage convention is a contract between the compiler and any outside callers of the compiled code. It creates a known interface that can be used to invoke a procedure and obtain any results that it returns (while protecting the caller's runtime environment). Thus, the compiler should only violate the linkage convention when such a violation cannot be detected from outside the compiled code.

a. Under what circumstances can the compiler be certain that using a variant linkage is safe? Give examples from real programming languages.

b. In these circumstances, what might the compiler change about the calling sequence and the linkage convention?