Contents

2 Programming Language Syntax

2.3.5 Recovering from Syntax Errors

EXAMPLE 2.43
Syntax error in C (reprise)

The main text illustrated the problem of syntax error recovery with a simple example in C:

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

The compiler will detect a syntax error immediately after the B, but it cannot give up at that point: it needs to keep looking for errors in the remainder of the program. To permit this, we must modify the input program, the state of the parser, or both, in a way that allows parsing to continue, hopefully without announcing a significant number of spurious cascading errors and without missing a significant number of real errors. The techniques discussed below allow the compiler to search for further syntax errors. In Chapter 4 we will consider additional techniques that allow it to search for additional static semantic errors as well. ■

Panic Mode

Perhaps the simplest form of syntax error recovery is a technique known as *panic mode*. It defines a small set of "safe symbols" that delimit clean points in the input. When an error occurs, a panic mode recovery algorithm deletes input tokens until it finds a safe symbol, then backs the parser out to a context in which that symbol might appear. In the earlier example, a recursive descent parser with panic mode recovery might delete input tokens until it finds the semicolon, return from all subroutines called from within stmt, and restart the body of stmt itself.

Unfortunately, panic mode tends to be a bit drastic. By limiting itself to a static set of "safe" symbols at which to resume parsing, it admits the possibility of deleting a significant amount of input while looking for such a symbol. Worse, if some of the deleted tokens are "starter" symbols that begin large-scale constructs in the language (e.g., begin, procedure, while), we shall almost surely see spurious cascading errors when we reach the end of the construct.

EXAMPLE 2.44
The problem with panic mode

Consider the following fragment of code in an Algol-family language:

```
IF a b THEN x;
ELSE y;
END;
```

When it discovers the error at `b` in the first line, a panic-mode recovery algorithm is likely to skip forward to the semicolon, thereby missing the `THEN`. When the parser finds the `ELSE` on line 2 it will produce a spurious error message. When it finds the `END` on line 3 it will think it has reached the end of the enclosing structure (e.g., the whole subroutine), and will probably generate additional cascading errors on subsequent lines. Panic mode tends to work acceptably only in relatively "unstructured" languages, such as Basic and (early) Fortran, which don't have many "starter" symbols. ■

Phrase-Level Recovery

We can improve the quality of recovery by employing different sets of "safe" symbols in different contexts. Parsers that incorporate this improvement are said to implement *phrase-level recovery*. When it discovers an error in an expression, for example, a phrase-level recovery algorithm can delete input tokens until it reaches something that is likely to follow an expression—or perhaps that is able to start an expression, in the hope that what was deleted was an ignorable prefix. This more local recovery is better than always backing out to the end of the current statement, because it gives us the opportunity to examine the parts of the statement (and maybe even the expression) that follow the erroneous tokens.

EXAMPLE 2.45
Phrase-level recovery in recursive descent

Niklaus Wirth, the inventor of Pascal, published an elegant implementation of phrase-level recovery for recursive descent parsers in 1976 [Wir76, Sec. 5.9]. The simplest version of his algorithm depends on the FIRST and FOLLOW sets defined at the end of Section 2.3.1. If the parsing routine for nonterminal *foo* discovers an error at the beginning of its code, it deletes incoming tokens until it finds a member of FIRST(*foo*), in which case it proceeds, or a member of FOLLOW(*foo*), in which case it returns:

```
procedure foo()
    if not (input_token ∈ FIRST(foo) or (EPS(foo) and input_token ∈ FOLLOW(foo))
        report_error()              –– print message for the user
        repeat
            delete_token()
        until input_token ∈ (FIRST(foo) ∪ FOLLOW(foo) ∪ {$$})
    case input_token of
        . . . : . . .
        . . . : . . .               –– valid starting tokens
        . . . : . . .
        otherwise return            –– error or foo ⟶ ε
```

Note that the report_error routine does *not* terminate the parse; it simply prints a message and returns. To complete the algorithm, the match routine must be altered so that it, too, will return after announcing an error, effectively inserting the expected token when something else appears:

```
procedure match(expected)
    if input_token = expected
        consume_input_token()
    else
        report_error()
```

Finally, to simplify the code, the common prefix of the various nonterminal subroutines can be moved into an error-checking subroutine:

```
procedure check_for_error(sym)
    if not (input_token ∈ FIRST(sym) or EPS(sym) and input_token ∈ FOLLOW(sym))
        report_error()
        repeat
            delete_token()
        until input_token ∈ (FIRST(sym) ∪ FOLLOW(sym) ∪ {$$})
```

■

Context-Specific Look-Ahead

Though simple, the recovery algorithm just described has an unfortunate tendency, when *foo* $\longrightarrow \epsilon$, to predict one or more epsilon productions when it should really announce an error right away. This weakness is known as the *immediate error detection* problem. It stems from the fact that FOLLOW(*foo*) is context-independent: it contains all tokens that may follow *foo* somewhere in some valid program, but not necessarily in the current context in the current program. This is basically the same observation that underlies the distinction between SLR and LALR parsers ("The Characteristic Finite-State Machine and LR Parsing Variants," Section 2.3.4).

EXAMPLE 2.46
Cascading syntax errors

As an example, consider the following incorrect code in our calculator language:

```
Y := (A * X X*X) + (B * X*X) + (C * X) + D
```

To a human being, it is pretty clear that the programmer forgot a `*` in the x^3 term of a polynomial. The recovery algorithm isn't so smart. In a recursive descent parser it will see an identifier (`X`) coming up on the input when it is inside the following routines:

```
program
stmt_list
stmt
expr
term
factor
expr
term
factor_tail
factor_tail
```

Since an `id` can follow a *factor_tail* in some programs (e.g., `A := B     C := D`), the innermost parsing routine will predict *factor_tail* $\longrightarrow \epsilon$, and simply return. At that point both the outer factor_tail and the inner term will be at the end of their code, and they, too, will return. Next, the inner expr will call term_tail, which will also predict an epsilon production, since an `id` can follow a term_tail in certain programs. This will leave the inner expr at the end of its code, allowing it to return. Only then will we discover an error, when factor calls match, expecting to see a right parenthesis. Afterward there will be a host of cascading errors, as the input is transformed into

```
Y := (A * X)
X := X
B := X*X
C := X
```

■

EXAMPLE 2.47

Reducing cascading errors with context-specific look-ahead

To avoid inappropriate epsilon predictions, Wirth introduced the notion of context-specific FOLLOW sets, passed into each nonterminal subroutine as an explicit parameter. In our example, we would pass `id` as part of the FOLLOW set for the initial, outer expr, which is called as part of the production *stmt* ⟶ `id :=` *expr*, but *not* into the second, inner expr, which is called as part of the production *factor* ⟶ `(` *expr* `)`. The nested calls to term and factor_tail will end up being called with a FOLLOW set whose only member is a right parenthesis. When the inner call to factor_tail discovers that `id` is not in FIRST(*factor_tail*), it will delete tokens up to the right parenthesis before returning. The net result is a single error message, and a transformation of the input into

```
Y := (A * X*X) + (B * X*X) + (C * X) + D
```

That's still not the "right" interpretation, but it's a lot better than it was. ■

The final version of Wirth's phrase-level recovery employs one additional heuristic: to avoid cascading errors it refrains from deleting members of a statically defined set of "starter" symbols (e.g., `begin`, `procedure`, `(`, etc.). These are the symbols that tend to require matching tokens later in the program. If we see a starter symbol while deleting input, we give up on the attempt to delete the rest of the erroneous construct. We simply return, even though we know that the starter symbol will not be acceptable to the calling routine. With context-specific FOLLOW sets and starter symbols, phrase-level recovery looks like this:

EXAMPLE 2.48

Recursive descent with full phrase-level recovery

```
procedure check_for_error(sym, follow_set)
    if not (input_token ∈ FIRST(sym) or (EPS(sym) and input_token ∈ follow_set))
        report_error()
        repeat
            delete_token()
        until input_token ∈ FIRST(sym) ∪ follow_set ∪ starter_set ∪ {$$}
```

```
procedure expr(follow_set)
    check_for_error(expr, follow_set)
    case input_token of
        . . . : . . .
        . . . : . . .                    valid starting tokens
        . . . : . . .
        otherwise return
```

Context-specific FOLLOW sets are tracked dynamically during the parse of a given input. Initially, in the augmenting production $S \longrightarrow$ *program* $$, the context-specific FOLLOW set for *program* is {$$}. Thus when calling the recursive descent routine for program, we pass {$$} as parameter. Then, within each routine, we determine the FOLLOW sets to pass to other routines based on whatever comes next in the current right-hand side, potentially augmented by what was already passed as the FOLLOW set of the current left-hand side. Specifically, suppose we are currently executing the recursive descent routine for symbol A, called with context-specific FOLLOW set S. Suppose further that we have realized (predicted) that we are in the production $A \longrightarrow \alpha\ B\ \beta$ and we are about to call the routine for symbol B. If $\beta \Longrightarrow^* \epsilon$, we will pass FIRST($\beta$) $\cup$ S as the context-specific FOLLOW set for B. If β cannot generate ϵ, we will simply pass FIRST(β).

Exception-Based Recovery in Recursive Descent

An attractive alternative to Wirth's technique relies on the exception-handling mechanisms available in many modern languages (we will discuss these mechanisms in detail in Section 9.4). Rather than implement recovery for every nonterminal in the language (a somewhat tedious task), the exception-based approach identifies a small set of contexts to which we back out in the event of an error. In many languages, we could obtain simple, but probably serviceable error recovery by backing out to the nearest statement or declaration. In the limit, if we choose a single place to "back out to," we have an implementation of panic-mode recovery.

EXAMPLE 2.49
Exceptions in a recursive descent parser

The basic idea is to attach an exception handler (a special syntactic construct) to the blocks of code in which we want to implement recovery:

```
procedure statement()
    try
        . . .                       –– code to parse a statement
    except when syntax_error
        loop
            if next_token $\in$ FIRST(statement)
                statement()         –– try again
                return
            elsif next_token $\in$ FOLLOW(statement)
                return
            else delete_token()
```

Code for declaration would be similar. For better-quality repair, we might add handlers around the bodies of expression, aggregate, or other complex constructs.

To guarantee that we can always recover from an error, we must ensure that all parts of the grammar lie inside at least one handler.

When we detect an error (possibly nested many procedure calls deep), we *raise* a syntax error exception ("`raise`" is a built-in command in languages with exceptions). The language implementation then unwinds the stack to the most recent context in which we have an exception handler, which it executes in place of the remainder of the block to which the handler is attached. For phrase-level (or panic mode) recovery, the handler can delete input tokens until it sees one with which it can recommence parsing.

As noted in Section 2.3.1, the ANTLR parser generator takes a CFG as input and builds a human-readable recursive descent parser. Compiler writers have the option of generating Java, C#, or C++, all of which have exception-handling mechanisms. When an ANTLR-generated parser encounters a syntax error, it throws a `MismatchedTokenException` or `NoViableAltException`. By default ANTLR includes a handler for these exceptions in every nonterminal subroutine. The handler prints an error message, deletes tokens until it finds something in the FOLLOW set of the nonterminal, and then returns. The compiler writer can define alternative handlers if desired on a production-by-production basis.

Error Productions

As a general rule, it is desirable for an error recovery technique to be as language-independent as possible. Even in a recursive descent parser, which is handwritten for a particular language, it is nice to be able to encapsulate error recovery in the check_for_error and match subroutines. Sometimes, however, one can obtain much better repairs by being highly language specific.

EXAMPLE 2.50
Error production for "`; else`"

Most languages have a few unintuitive rules that programmers tend to violate in predictable ways. In Pascal, for example, semicolons are used to separate statements, but many programmers think of them as *terminating* statements instead. Most of the time the difference is unimportant, since a statement is allowed to be empty. In the following, for example,

```
begin
    x := (-b + sqrt(b*b -4*a*c)) / (2*a);
    writeln(x);
end;
```

the compiler parses the `begin`... `end` block as a sequence of three statements, the third of which is empty. In the following, however,

```
if d <> 0 then
    a := n/d;
else
    a := n;
end;
```

the compiler must complain, since the `then` part of an `if`... `then` ... `else` construct must consist of a single statement in Pascal. A Pascal semicolon is never allowed immediately before an `else`, but programmers put them there all the time. Rather than try to tune a general recovery or repair algorithm to deal correctly with this problem, most Pascal compiler writers modify the grammar: they include an extra production that allows the semicolon, but causes the semantic analyzer to print a warning message, telling the user that the semicolon shouldn't be there. Similar error productions are used in C compilers to cope with "anachronisms" that have crept into the language as it evolved. Syntax that was valid only in early versions of C is still accepted by the parser, but evokes a warning message. ■

Error Recovery in Table-Driven LL Parsers

Given the similarity to recursive descent parsing, it is straightforward to implement phrase-level recovery in a table-driven top-down parser. Whenever we encounter an error entry in the parse table, we simply delete input tokens until we find a member of a statically defined set of starter symbols (including `$$`), or a member of the FIRST or FOLLOW set of the nonterminal at the top of the parse stack.[1] If we find a member of the FIRST set, we continue the main loop of the driver. If we find a member of the FOLLOW set or the starter set, we pop the nonterminal off the parse stack first. If we encounter an error in `match`, rather than in the parse table, we simply pop the token off the parse stack.

But we can do better than this! Since we have the entire parse stack easily accessible (it was hidden in the control flow and procedure calling sequence of recursive descent), we can enumerate all possible combinations of insertions and deletions that would allow us to continue parsing. Given appropriate metrics, we can then evaluate the alternatives to pick the one that is in some sense "best."

Because perfect error recovery (actually error *repair*) would require that we read the programmer's mind, any practical technique to evaluate alternative "corrections" must rely on heuristics. For the sake of simplicity, most compilers limit themselves to heuristics that (1) require no semantic information, (2) do not require that we "back up" the parser or the input stream (i.e., to some state prior to the one in which the error was detected), and (3) do not change the spelling of tokens or the boundaries between them. A particularly elegant algorithm that conforms to these limits was published by Fischer, Milton, and Quiring in 1980 [FMQ80]. As originally described, the algorithm was limited to languages in which programs could always be corrected by inserting appropriate tokens into the input stream, without ever requiring deletions. It is relatively easy, however, to extend the algorithm to encompass deletions and substitutions. We consider the insert-only algorithm first; the version with deletions employs it as a subroutine. We do not consider substitutions here.[2]

[1] This description uses global FOLLOW sets. If we want to use context-specific look-aheads instead, we can peek farther down in the stack. A token is an acceptable context-specific look-ahead if it is in the FIRST set of the second symbol A from the top in the stack or, if it would cause us to predict $A \longrightarrow \varepsilon$, the FIRST set of the third symbol B from the top or, if it would cause us to predict $B \longrightarrow \varepsilon$, the FIRST set of the fourth symbol from the top, and so on.

The FMQ error-repair algorithm requires the compiler writer to assign an insertion cost $C(\mathtt{t})$ and a deletion cost $D(\mathtt{t})$ to every token t. (Since we cannot change where the input ends, we have $C(\mathtt{\$\$}) = D(\mathtt{\$\$}) = \infty$.) In any given error situation, the algorithm chooses the least cost combination of insertions and deletions that allows the parser to consume one more token of real input. The state of the parser is never changed; only the input is modified (rather than pop a stack symbol, the repair algorithm pushes its yield onto the input stream).

As in phrase-level recovery in a recursive descent parser, the FMQ algorithm needs to address the immediate error detection problem. There are several ways we could do this. One would be to use a "full LL" parser, which keeps track of local FOLLOW sets. Another would be to inspect the stack when predicting an epsilon production, to see if what lies underneath will allow us to accept the incoming token. The first option significantly increases the size and complexity of the parser. The second option leads to a nonlinear-time parsing algorithm. Fortunately, there is a third option. We can save all changes to the stack (and calls to the semantic analyzer's action routines) in a temporary buffer until the match routine accepts another real token of input. If we discover an error before we accept a real token, we undo the stack changes and throw away the buffered calls to action routines. Then we can pretend we recognized the error when a full LL parser would have.

We now consider the task of repairing with only insertions. We begin by extending the notion of insertion costs to strings in the obvious way: if $w = a_1 a_2 \ldots a_n$, we have $C(w) = \sum_{i=1}^{n} C(a_i)$. Using the cost function C, we then build a pair of tables S and E. The S table is one-dimensional, and is indexed by grammar symbol. For any symbol X, $S(X)$ is a least-cost string of terminals derivable from X. That is,

$$S(X) = w \iff X \Longrightarrow^* w \text{ and } \forall x \text{ such that } X \Longrightarrow^* x,\ C(w) \leq C(x)$$

Clearly $S(\mathtt{a}) = \mathtt{a}\ \forall$ tokens a.

The E table is two-dimensional, and is indexed by symbol/token pairs. For any symbol X and token a, $E(X, \mathtt{a})$ is the lowest-cost prefix of a in X; that is, the lowest cost token string w such that $X \Longrightarrow^* w\mathtt{a}x$. If X cannot yield a string containing a, then $E(X, \mathtt{a})$ is defined to be a special symbol ?? whose insertion cost is ∞. If X = a, or if $X \Longrightarrow^* \mathtt{a}x$, then $E(X, \mathtt{a}) = \varepsilon$, where $C(\varepsilon) = 0$.

EXAMPLE 2.51
Insertion-only repair in FMQ

To find a least-cost insertion that will repair a given error, we execute the function find_insertion, shown in Figure C-2.31. The function begins by considering the least-cost insertion that will allow it to derive the input token from the symbol at the top of the stack (there may be none). It then considers the possibility of "deleting" that top-of-stack symbol (by inserting its least-cost yield into the input stream) and deriving the input token from the second symbol on the stack. It

[2] A substitution can always be effected as a deletion/insertion pair, but we might want ideally to give it special consideration. For example, we probably want to be cautious about deleting a left square bracket or inserting a left parenthesis, since both of these symbols must be matched by something later in the input, at which point we are likely to see cascading errors. But substituting a left parenthesis for a left square bracket is in some sense more plausible, especially given the differences in array subscript syntax in different programming languages.

```
function find_insertion(a : token) : string
    –– assume that the parse stack consists of symbols $X_n, \ldots X_2, X_1$,
    –– with $X_n$ at top-of-stack
    ins := ??
    prefix := $\epsilon$
    for $i$ in $n$ . . 1
        if $C$(prefix) $\geq$ $C$(ins)
            –– no better insertion is possible
            return ins
        if $C$(prefix . $E(X_i$, a)) < $C$(ins)
            –– better insertion found
            ins := prefix . $E(X_i$, a)
        prefix := prefix . $S(X_i)$
    return ins
```

Figure 2.31 **Outline of a function to find a least-cost insertion that will allow the parser to accept the input token** a. The dot character (.) is used here for string concatenation.

```
function find_repair() : ⟨string, int⟩
    –– assume that the parse stack consists of symbols $X_n, \ldots X_2, X_1$,
    –– with $X_n$ at top-of-stack,
    –– and that the input stream consists of tokens $a_1, a_2, a_3, \ldots$
    i := 0        –– number of tokens we're considering deleting
    best_ins := ??
    best_del := 0
    loop
        cur_ins := find_insertion($a_{i+1}$)
        if $C$(cur_ins) + $D(a_1 \ldots a_i)$ < $C$(best_ins) + $D(a_1 \ldots a_{\text{best\_del}})$
            –– better repair found
            best_ins := cur_ins
            best_del := i
        i +:= 1
        if $D(a_1 \ldots a_i)$ > $C$(best_ins) + $D(a_1 \ldots a_{\text{best\_del}})$
            –– no better repair is possible
            return ⟨best_ins, best_del⟩
```

Figure 2.32 **Outline of a function to find a least-cost combination of insertions and deletions that will allow the parser to accept one more token of input.**

continues in this fashion, considering ways to derive the input token from ever deeper symbols on the stack, until the cost of inserting the yields of the symbols above exceeds the cost of the cheapest repair found so far. If it reaches the bottom of the stack without finding a finite-cost repair, then the error cannot be repaired by insertions alone.

EXAMPLE 2.52
FMQ with deletions

To produce better-quality repairs, and to handle languages that cannot be repaired with insertions only, we need to consider deletions. As we did with the insert cost vector C, we extend the deletion cost vector D to strings of tokens in the obvious way. We then embed calls to find_insertion in a second loop, shown in Figure C-2.32. This loop repeatedly considers deleting more and more tokens, each time calling find_insertion on the remaining input, until the cost of deleting additional tokens exceeds the cost of the cheapest repair found so far. The search can never fail; it is always possible to find a combination of insertions and deletions that will allow the end-of-file token to be accepted. Since the algorithm may need to consider (and then reject) the option of deleting an arbitrary number of tokens, the scanner must be prepared to peek an arbitrary distance ahead in the input stream and then back up again. ■

The FMQ algorithm has several desirable properties. It is simple and efficient (given that the grammar is bounded in size, we can prove that the time to choose a repair is bounded by a constant). It can repair an arbitrary input string. Its decisions are locally optimal, in the sense that no cheaper repair can allow the parser to make forward progress. It is table-driven and therefore fully automatic. Finally, it can be tuned to prefer "likely" repairs by modifying the insertion and deletion costs of tokens. Some obvious heuristics include:

- Deletion should usually be more expensive than insertion.
- Common operators (e.g., multiplication) should have lower cost than uncommon operators (e.g., modulo division) in the same place in the grammar.
- Starter symbols (e.g., `begin`, `if`, `(`) should have higher cost than their corresponding final symbols (`end`, `fi`, `)`).
- "Noise" symbols (comma, semicolon, `do`) should have very low cost.

Error Recovery in Bottom-Up Parsers

Locally least-cost repair is possible in bottom-up parsers, but it isn't as easy as it is in top-down parsers. The advantage of a top-down parser is that the content of the parse stack unambiguously identifies the context of an error, and specifies the constructs expected in the future. The stack of a bottom-up parser, by contrast, describes a set of possible contexts, and says nothing explicit about the future.

In practice, most bottom-up parsers tend to rely on panic-mode or phrase-level recovery. The intuition is that when an error occurs, the top few states on the parse stack represent the shifted prefix of an erroneous construct. Recovery consists of popping these states off the stack, deleting the remainder of the construct from the incoming token stream, and then restarting the parser, possibly after shifting a fictitious nonterminal to represent the erroneous construct.

Unix's `yacc/bison` provides a typical example of bottom-up phrase-level recovery. In addition to the usual tokens of the language, `yacc/bison` allows the compiler writer to include a special token, `error`, anywhere in the right-hand sides of grammar productions. When the parser built from the grammar detects a syntax error, it

1. Calls the function `yyerror`, which the compiler writer must provide. Normally, `yyerror` simply prints a message (e.g., "parse error"), which `yacc/bison` passes as an argument

2. Pops states off the parse stack until it finds a state in which it can shift the `error` token (if there is no such state, the parser terminates)
3. Inserts and then shifts the `error` token
4. Deletes tokens from the input stream until it finds a valid look-ahead for the new (post `error`) context
5. Temporarily disables reporting of further errors
6. Resumes parsing

If there are any semantic action routines associated with the production containing the `error` token, these are executed in the normal fashion. They can do such things as print additional error messages, modify the symbol table, patch up semantic processing, prompt the user for additional input in an interactive tool (`yacc/bison` can be used to build things other than batch-mode compilers), or disable code generation. The rationale for disabling further syntax errors is to make sure that we have really found an acceptable context in which to resume parsing before risking cascading errors. `Yacc/bison` automatically reenables the reporting of errors after successfully shifting three real tokens of input. A semantic action routine can reenable error messages sooner if desired by calling the built-in routine `yyerrorok`.

EXAMPLE 2.53
Panic mode in `yacc/bison`

For our example calculator language, we can imagine building a `yacc/bison` parser using the bottom-up grammar of Figure 2.25. For panic-mode recovery, we might want to back out to the nearest statement:

```
stmt  ⟶  error
         {printf("parsing resumed at end of current statement\n");}
```

The semantic routine written in curly braces would be executed when the parser recognizes *stmt* ⟶ `error`.[3] Parsing would resume at the next token that can follow a statement—in our calculator language, at the next `id`, `read`, `write`, or `$$`. ■

EXAMPLE 2.54
Panic mode with statement terminators

A weakness of the calculator language, from the point of view of error recovery, is that the current, erroneous statement may well contain additional `id`s. If we resume parsing at one of these, we are likely to see another error right away. We could avoid the error by disabling error messages until several real tokens have been shifted. In a language in which every statement ends with a semicolon, we could have more safely written

```
stmt  ⟶  error ;
         {printf("parsing resumed at end of current statement\n");}
```

■

EXAMPLE 2.55
Phrase-level recovery in `yacc/bison`

In both of these examples we have placed the `error` symbol at the beginning of a right-hand side, but there is no rule that says it must be so. We might decide,

[3] The syntax shown here is not the same as that accepted by `yacc/bison`, but is used for the sake of consistency with earlier material.

for example, that we will abandon the current statement whenever we see an error, unless the error happens inside a parenthesized expression, in which case we will attempt to resume parsing after the closing parenthesis. We could then add the following production:

```
factor ⟶ ( error )
           {printf("parsing resumed at end of parenthesized expression\n");}
```

In the CFSM of Figure 2.26, it would then be possible in State 8 to shift `error`, delete some tokens, shift `)`, recognize *factor*, and continue parsing the surrounding expression. Of course, if the erroneous expression contains nested parentheses, the parser may not skip all of it, and a cascading error may still occur. ■

Because `yacc/bison` creates LALR parsers, it automatically employs context-specific look-ahead, and does not usually suffer from the immediate error detection problem. (A full LR parser would do slightly better.) In an SLR parser, a good error recovery algorithm needs to employ the same trick we used in the top-down case. Specifically, we buffer all stack changes and calls to semantic action routines until the shift routine accepts a real token of input. If we discover an error before we accept a real token, we undo the stack changes and throw away the buffered calls to semantic routines. Then we can pretend we recognized the error when a full LR parser would have.

✓ CHECK YOUR UNDERSTANDING

44. Why is syntax error recovery important?
45. What are *cascading errors*?
46. What is *panic mode*? What is its principal weakness?
47. What is the advantage of *phrase-level recovery* over panic mode?
48. What is the *immediate error detection problem*, and how can it be addressed?
49. Describe two situations in which context-specific FOLLOW sets may be useful.
50. Outline Wirth's mechanism for error recovery in recursive descent parsers. Compare this mechanism to exception-based recovery.
51. What are *error productions*? Why might a parser that incorporates a high-quality, general-purpose error recovery algorithm still benefit from using such productions?
52. Outline the FMQ algorithm. In what sense is the algorithm optimal?
53. Why is error recovery more difficult in bottom-up parsers than it is in top-down parsers?
54. Describe the error recovery mechanism employed by `yacc/bison`.

2 Programming Language Syntax

2.4 Theoretical Foundations

As noted in the main text, scanners and parsers are based on the finite automata and pushdown automata that form the bottom two levels of the Chomsky language hierarchy. At each level of the hierarchy, machines can be either *deterministic* or *nondeterministic*. A deterministic automaton always performs the same operation in a given situation. A nondeterministic automaton can perform any of a *set* of operations. A nondeterministic machine is said to accept a string if there exists a choice of operation in each situation that will eventually lead to the machine saying "yes." As it turns out, nondeterministic and deterministic finite automata are equally powerful: every DFA is, by definition, a degenerate NFA, and the construction in Example 2.14 demonstrates that for any NFA we can create a DFA that accepts the same language. The same is not true of push-down automata: there are context-free languages that are accepted by an NPDA but not by any DPDA. Fortunately, DPDAs suffice in practice to accept the syntax of real programming languages. Practical scanners and parsers are always deterministic.

2.4.1 Finite Automata

Precisely defined, a deterministic finite automaton (DFA) M consists of (1) a finite set Q of *states*, (2) a finite alphabet Σ of input symbols, (3) a distinguished *initial* state $q_1 \in Q$, (4) a set of distinguished *final* states $F \subseteq Q$, and (5) a *transition function* $\delta : Q \times \Sigma \rightarrow Q$ that chooses a new state for M based on the current state and the current input symbol. M begins in state q_1. One by one it consumes its input symbols, using δ to move from state to state. When the final symbol has been consumed, M is interpreted as saying "yes" if it is in a state in F; otherwise it is interpreted as saying "no." We can extend δ in the obvious way to take strings, rather than symbols, as inputs, allowing us to say that M accepts string x if $\delta(q_1, x) \in F$. We can then define $L(M)$, the language accepted by M,

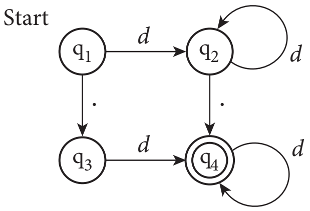


Figure 2.33 **Minimal DFA for the language consisting of all strings of decimal digits containing a single decimal point.** Adapted from Figure 2.10 in the main text. The symbol d here is short for "0, 1, 2, 3, 4, 5, 6, 7, 8, 9".

to be the set $\{x \mid \delta(q_1, x) \in F\}$. In a nondeterministic finite automaton (NFA), the transition function, δ, is multivalued: the automaton can move to any of a *set* of possible states from a given state on a given input. In addition, it may move from one state to another "spontaneously"; such transitions are said to take input symbol ε.

EXAMPLE 2.56

Formal DFA for d*(. d | d .) d*

We can illustrate these definitions with an example. Consider the circles-and-arrows automaton of Figure C-2.33 (adapted from Figure 2.10 in the main text). This is the minimal DFA accepting strings of decimal digits containing a single decimal point. $\Sigma = \{\texttt{0}, \texttt{1}, \texttt{2}, \texttt{3}, \texttt{4}, \texttt{5}, \texttt{6}, \texttt{7}, \texttt{8}, \texttt{9}, \texttt{.}\}$ is the machine's input alphabet. $Q = \{q_1, q_2, q_3, q_4\}$ is the set of states; q_1 is the initial state; $F = \{q_4\}$ (a singleton in this case) is the set of final states. The transition function can be represented by a set of triples $\delta = \{(q_1, \texttt{0}, q_2), \ldots, (q_1, \texttt{9}, q_2), (q_1, \texttt{.}, q_3), (q_2, \texttt{0}, q_2), \ldots, (q_2, \texttt{9}, q_2), (q_2, \texttt{.}, q_4), (q_3, \texttt{0}, q_4), \ldots, (q_3, \texttt{9}, q_4), (q_4, \texttt{0}, q_4), \ldots, (q_4, \texttt{9}, q_4)\}$. In each triple $(q_i, \texttt{a}, q_j)$, $\delta(q_i, \texttt{a}) = q_j$. ■

Given the constructions of Examples 2.12 and 2.14, we know that there exists an NFA that accepts the language generated by any given regular expression, and a DFA equivalent to any given NFA. To show that regular expressions and finite automata are of equivalent expressive power, all that remains is to demonstrate that there exists a regular expression that generates the language accepted by any given DFA. We illustrate the required construction below for our decimal strings example (Figure C-2.33). More formal and general treatment of all the regular language constructions can be found in standard automata theory texts [HMU07, Sip13].

From a DFA to a Regular Expression

To construct a regular expression equivalent to a given DFA, we employ a dynamic programming algorithm that builds solutions to successively more complicated subproblems from a table of solutions to simpler subproblems. We begin with a set of simple regular expressions that describe the transition function, δ. For all states i, we define

$$r^0_{ii} = \texttt{a}_1 \mid \texttt{a}_2 \mid \ldots \mid \texttt{a}_m \mid \varepsilon$$

where $\{\texttt{a}_1 \mid \texttt{a}_2 \mid \ldots \mid \texttt{a}_m\} = \{\texttt{a} \mid \delta(q_i, \texttt{a}) = q_i\}$ is the set of characters labeling the "self-loop" from state q_i back to itself. If there is no such self-loop, $r^0_{ij} = \varepsilon$. Similarly, for $i \neq j$, we define

$$r^0_{ij} = \texttt{a}_1 \mid \texttt{a}_2 \mid \ldots \mid \texttt{a}_m$$

where $\{\texttt{a}_1 \mid \texttt{a}_2 \mid \ldots \mid \texttt{a}_m\} = \{\texttt{a} \mid \delta(q_i, \texttt{a}) = q_j\}$ is the set of characters labeling the arc from q_i to q_j. If there is no such arc, r^0_{ij} is the empty regular expression. (Note the difference here: we can stay in state q_i by not accepting any input, so ε is always one of the alternatives in r^0_{ii}, but not in r^0_{ij} when $i \neq j$.)

Given these r^0 expressions, the dynamic programming algorithm inductively calculates expressions r^k_{ij} with larger superscripts. In each, k names the highest-numbered state through which control may pass on the way from q_i to q_j. We assume that states are numbered starting with q_1, so when $k = 0$ we must transition directly from q_i to q_j, with no intervening states.

EXAMPLE 2.57

Reconstructing a regular expression for the decimal string DFA

In our small example DFA, $r^0_{11} = r^0_{33} = \varepsilon$, and $r^0_{22} = r^0_{44} = \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \mid \varepsilon$, which we will abbreviate $d \mid \varepsilon$. Similarly, $r^0_{13} = r^0_{24} = \texttt{.}$, and $r^0_{12} = r^0_{34} = d$. Expressions r^0_{14}, r^0_{21}, r^0_{23}, r^0_{31}, r^0_{32}, r^0_{41}, r^0_{42}, and r^0_{43} are all empty.

For $k > 0$, the r^k_{ij} expressions will generally generate multicharacter strings. At each step of the dynamic programming algorithm, we let

$$r^k_{ij} = r^{k-1}_{ij} \mid r^{k-1}_{ik} r^{k-1}_{kk}{}^{*} r^{k-1}_{kj}$$

That is, to get from q_i to q_j without going through any states numbered higher than k, we can either go from q_i to q_j without going through any state numbered higher than $k - 1$ (which we already know how to do), or else we can go from q_i to q_k (without going through any state numbered higher than $k - 1$), travel out from q_k and back again an arbitrary number of times (never visiting a state numbered higher than $k - 1$ in between), and finally go from q_k to q_j (again without visiting a state numbered higher than $k - 1$). If any of the constituent regular expressions is empty, we omit its term of the outermost alternation. At the end, our overall answer is $r^n_{1f_1} \mid r^n_{1f_2} \mid \ldots \mid r^n_{1f_t}$, where $n = |Q|$ is the total number of states and $F = \{q_{f_1}, q_{f_2}, \ldots, q_{f_t}\}$ is the set of final states.

Because $r^0_{11} = \varepsilon$ and there are no transitions from States 2, 3, or 4 to State 1, nothing changes in the first inductive step in our example; that is, $\forall i\, [\, r^1_{ii} = r^0_{ii}\,]$. The second step is a bit more interesting. Since we are now allowed to go through State 2, we have $r^2_{22} = r^2_{22}\ r^2_{22}{}^{*} r^2_{22} = (\, d \mid \varepsilon\,) \mid (\, d \mid \varepsilon\,)\, (\, d \mid \varepsilon\,)^{*} (\, d \mid \varepsilon\,) = d^{*}$. Because r^1_{21}, r^1_{23}, r^1_{32}, and r^1_{42} are empty, however, r^2_{11}, r^2_{33}, and r^2_{44} remain the same as r^1_{11}, r^1_{33}, and r^1_{44}. In a similar vein, we have

$$\begin{aligned}
r^2_{12} &= d \mid d\,(\, d \mid \varepsilon\,)^{*}(\, d \mid \varepsilon\,) &&= d^{+} \\
r^2_{14} &= d\,(\, d \mid \varepsilon\,)^{*}\ . &&= d^{+}\ . \\
r^2_{24} &= \ .\ \mid (\, d \mid \varepsilon\,)\,(\, d \mid \varepsilon\,)^{*}\ . &&= d^{*}\ .
\end{aligned}$$

Missing transitions and empty expressions from the previous step leave $r_{13}^2 = r_{13}^1 = .$ and $r_{34}^2 = r_{34}^1 = d$. Expressions r_{21}^2, r_{23}^2, r_{31}^2, r_{32}^2, r_{41}^2, r_{42}^2, and r_{43}^2 remain empty.

In the third inductive step, we have

$$\begin{aligned} r_{13}^3 &= \ . \mid . \ \varepsilon^* \varepsilon \ = \ . \\ r_{14}^3 &= d^+ \ . \mid . \ \varepsilon^* d \ = \ d^+ \ . \mid . \ d \\ r_{34}^3 &= d \mid \varepsilon\varepsilon^* d \ = \ d \end{aligned}$$

All other expressions remain unchanged from the previous step.

Finally, we have

$$\begin{aligned} r_{14}^4 &= (\, d^+ \ . \mid . \ d \,) \mid (\, d^+ \ . \mid . \ d \,)\,(\, d \mid \varepsilon \,)^*(\, d \mid \varepsilon \,) \\ &= (\, d^+ \ . \mid . \ d \,) \mid (\, d^+ \ . \mid . \ d \,)\, d^* \\ &= (\, d^+ \ . \mid . \ d \,)\, d^* \\ &= d^+ \ . \ d^* \mid . \ d^+ \end{aligned}$$

Since F has a single member (q_4), this expression is our final answer.

Space Requirements

In Section 2.2.1 we noted without proof that the conversion from an NFA to a DFA may lead to exponential blow-up in the number of states. Certainly this did not happen in our decimal string example: the NFA of Figure 2.8 has 14 states, while the equivalent DFA of Figure 2.9 has only 7, and the minimal DFA of Figures 2.10 and C-2.33 has only 4.

EXAMPLE 2.58

A regular language with a large minimal DFA

Consider, however, the subset of `( a | b | c )*` in which some letter appears at least three times. The minimal DFA for this language has 28 states. As shown in Figure C-2.34, 27 of these are states in which we have seen i, j, and k `a`s, `b`s, and `c`s, respectively. The 28th (and only final) state is reached once we have seen at least three of some specific character.

By contrast, there exists an NFA for this language with only eight states, as shown in Figure C-2.35. It requires that we "guess," at the outset, whether we will see three `a`s, three `b`s, or three `c`s. It mirrors the structure of the natural regular expression `( a | b | c )* a ( a | b | c )* a ( a | b | c )* a ( a | b | c )* | ( a | b | c )* b ( a | b | c )* b ( a | b | c )* b ( a | b | c )* | ( a | b | c )* c ( a | b | c )* c ( a | b | c )* c ( a | b | c )*`.

EXAMPLE 2.59

Exponential DFA blow-up

Of course, the eight-state NFA does not emerge directly from the construction of Figure 2.7; that construction produces a 52-state machine with a certain amount of redundancy, and with many extraneous states and epsilon productions. But consider the similar subset of `( 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 )*` in which some digit appears at least ten times. The minimal DFA for this language has 10,000,000,001 states: a non-final state for each combination of zeros through nines with less than ten of each, and a single final state reached once any digit has appeared at least ten times. One possible regular expression for this language is

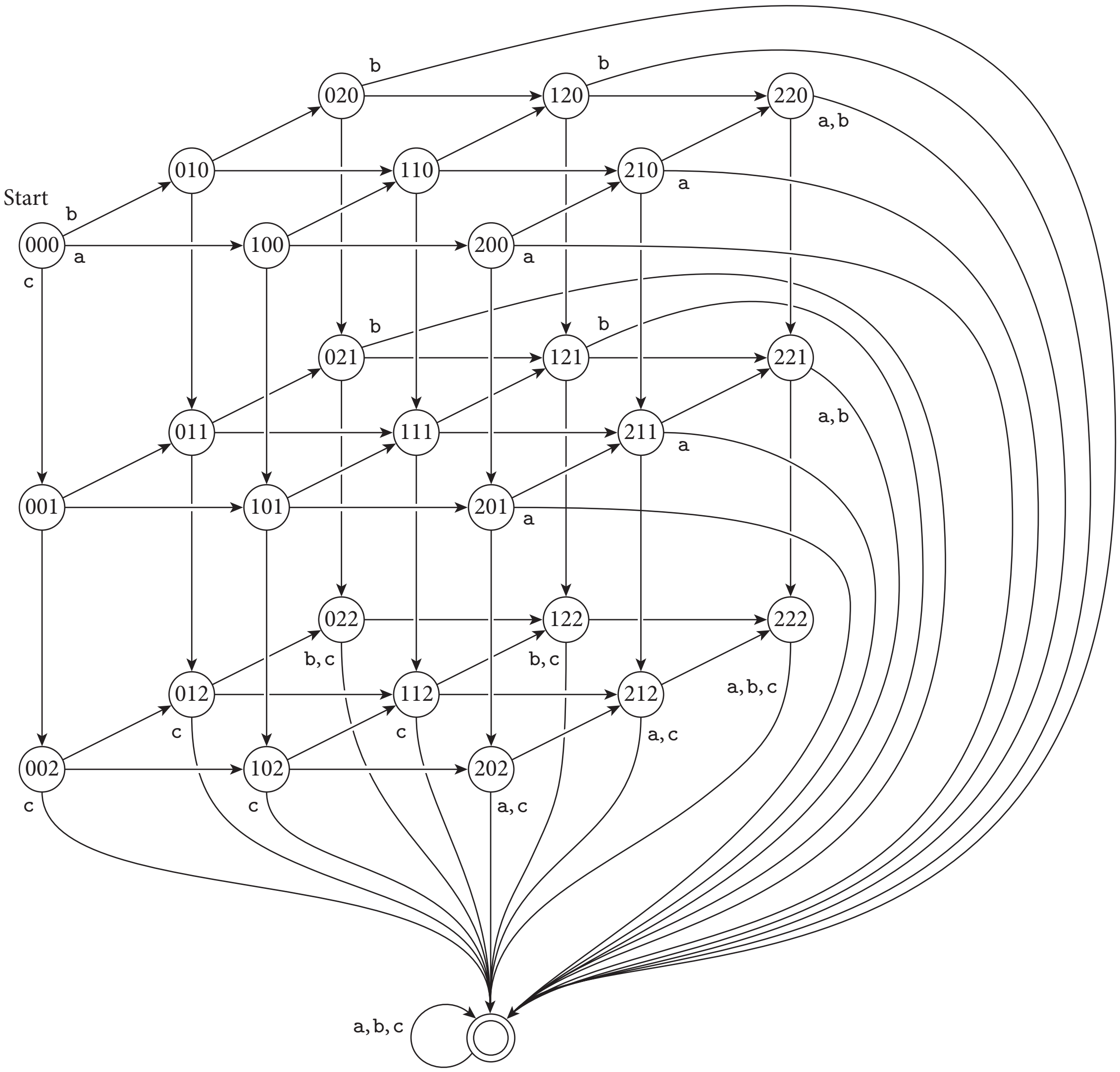


Figure 2.34 DFA for the language consisting of all strings in (a | b | c)* in which some letter appears at least three times. State name *ijk* indicates that *i* as, *j* bs, and *k* cs have been seen so far. Within the cubic portion of the figure, most edge labels are elided: a transitions move to the right, b transitions go back into the page, and c transitions move down.

```
 ((0|1|...|9)* 0 (0|1|...|9)* 0 (0|1|...|9)* 0 (0|1|...|9)* 0
  (0|1|...|9)* 0 (0|1|...|9)* 0 (0|1|...|9)* 0 (0|1|...|9)* 0
  (0|1|...|9)* 0 (0|1|...|9)* 0 (0|1|...|9)*)
| ((0|1|...|9)* 1 (0|1|...|9)* 1 (0|1|...|9)* 1 (0|1|...|9)* 1
  (0|1|...|9)* 1 (0|1|...|9)* 1 (0|1|...|9)* 1 (0|1|...|9)* 1
  (0|1|...|9)* 1 (0|1|...|9)* 1 (0|1|...|9)*)
| ...
```

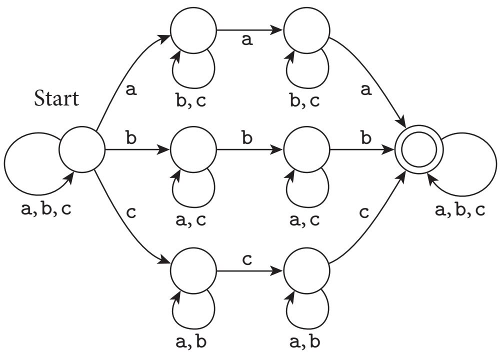


Figure 2.35 NFA corresponding to the DFA of Figure C-2.34.

```
| ((0|1|...|9)* 9 (0|1|...|9)* 9 (0|1|...|9)* 9 (0|1|...|9)* 9
   (0|1|...|9)* 9 (0|1|...|9)* 9 (0|1|...|9)* 9 (0|1|...|9)* 9
   (0|1|...|9)* 9 (0|1|...|9)* 9 (0|1|...|9)*)
```

Our construction would yield a very large NFA for this expression, but clearly many orders of magnitude smaller than ten billion states!

2.4.2 Push-Down Automata

A deterministic push-down automaton (DPDA) N consists of (1) Q, (2) Σ, (3) q_1, and (4) F, as in a DFA, plus (6) a finite alphabet Γ of stack symbols, (7) a distinguished initial stack symbol $Z_1 \in \Gamma$, and (5′) a transition function $\delta : Q \times \Gamma \times \{\Sigma \cup \{\varepsilon\}\} \rightarrow Q \times \Gamma^*$, where Γ^* is the set of strings of zero or more symbols from Γ. N begins in state q_1, with symbol Z_1 in an otherwise empty stack. It repeatedly examines the current state q and top-of-stack symbol Z. If $\delta(q,\varepsilon, Z)$ is defined, N moves to state r and replaces Z with α in the stack, where $(r, \alpha) = \delta(q,\varepsilon, Z)$. In this case N does not consume its input symbol. If $\delta(q,\varepsilon, Z)$ is undefined, N examines and consumes the current input symbol a. It then moves to state s and replaces Z with β, where $(s, \beta) = \delta(q, \mathtt{a}, Z)$. N is interpreted as accepting a string of input symbols if and only if it consumes the symbols and ends in a state in F.

As with finite automata, a nondeterministic push-down automaton (NPDA) is distinguished by a multivalued transition function: an NPDA can choose any of a set of new states and stack symbol replacements when faced with a given state, input, and top-of-stack symbol. If $\delta(q,\varepsilon, Z)$ is nonempty, N can also choose a new state and stack symbol replacement without inspecting or consuming its current input symbol. While we have seen that nondeterministic and deterministic finite automata are equally powerful, this correspondence does not carry over to push-down automata: there are context-free languages that are accepted by an NPDA but not by any DPDA.

The proof that CFGs and NPDAs are equivalent in expressive power is more complex than the corresponding proof for regular expressions and finite automata. The proof is also of limited practical importance for compiler construction; we do not present it here. While it is possible to create an NPDA for any CFL, simulating that NPDA may in some cases require exponential time to recognize strings in the language. (The $O(n^3)$ algorithms mentioned in Section 2.3 do not take the form of PDAs.) Practical programming languages can all be expressed with LL or LR grammars, which can be parsed with a (deterministic) PDA in linear time.

An LL(1) PDA is very simple. Because it makes decisions solely on the basis of the current input token and top-of-stack symbol, its state diagram is trivial. All but one of the transitions is a self-loop from the initial state to itself. A final transition moves from the initial state to a second, final state when it sees `$$` on the input and the stack. As we noted in Section 2.3.4, the state diagram for an SLR(1) or LALR(1) parser is substantially more interesting: it's the characteristic finite-state machine (CFSM). Full LR(1) parsers have similar machines, but usually with many more states, due to the need for path-specific look-ahead.

A little study reveals that if we define every state to be accepting, then the CFSM, without its stack, is a DFA that recognizes the grammar's *viable prefixes*. These are all the strings of grammar symbols that can begin a sentential form in the canonical (right-most) derivation of some string in the language, and that do not extend beyond the end of the handle. The algorithms to construct LL(1) and SLR(1) PDAs from suitable grammars were given in Sections 2.3.3 and 2.3.4.

2.4.3 Grammar and Language Classes

EXAMPLE 2.60

0^n1^n is not a regular language

As we noted in Section 2.1.2, a scanner is incapable of recognizing arbitrarily nested constructs. The key to the proof is to realize that we cannot count an arbitrary number of left-bracketing symbols with a finite number of states. Consider, for example, the problem of accepting the language 0^n1^n. Suppose there is a DFA M that accepts this language. Suppose further that M has m states. Now suppose we feed M a string of $m + 1$ zeros. By the *pigeonhole principle* (you can't distribute m objects among $p < m$ pigeonholes without putting at least two objects in some pigeonhole), M must enter some state q_i twice while scanning this string. Without loss of generality, let us assume it does so after seeing j zeros and again after seeing k zeros, for $j \neq k$. Since we know that M accepts the string 0^j1^j and the string 0^k1^k, and since it is in precisely the same state after reading 0^j and 0^k, we can deduce that M must also accept the strings 0^j1^k and 0^k1^j. Since these strings are not in the language, we have a contradiction: M cannot exist. ■

Within the family of context-free languages, one can prove similar theorems about the constructs that can and cannot be recognized using various parsing algorithms. Though almost all real parsers get by with a single token of look-ahead, it is possible in principle to use more than one, thereby expanding the set of grammars that can be parsed in linear time. In the top-down case we can redefine FIRST and FOLLOW sets to contain pairs of tokens in a more or less straightforward fashion. If we do this, however, we encounter a more serious version of the immediate error detection problem described in Section C-2.3.5. There we saw that the use of context-independent FOLLOW sets could cause us to overlook a syntax error until after we had needlessly predicted one or more epsilon productions. Context-specific FOLLOW sets solved the problem, but did not change the set of *valid* programs that could be parsed with one token of look-ahead. If we define LL(k) to be the set of all grammars that can be parsed predictively using the top-of-stack symbol and k tokens of look-ahead, then it turns out that for $k > 1$ we must adopt a context-specific notion of FOLLOW sets in order to parse correctly. The algorithm of Section 2.3.3, which is based on context-independent FOLLOW sets, is actually known as SLL (simple LL), rather than true LL. For $k = 1$, the LL(1) and SLL(1) algorithms can parse the same set of grammars. For $k > 1$, LL is strictly more powerful. Among the bottom-up parsers, the relationships among SLR(k), LALR(k), and LR(k) are somewhat more complicated, but extra look-ahead always helps.

EXAMPLE 2.61
Separation of grammar classes

Containment relationships among the classes of grammars accepted by popular linear-time algorithms appear in Figure C-2.36. The LR class (no suffix) contains every grammar G for which there exists a k such that $G \in$ LR(k); LL, SLL, SLR, and LALR are similarly defined. Grammars can be found in every region of the figure. Examples appear in Figure C-2.37. Proofs that they lie in the regions claimed are deferred to Exercise C-2.35.

For any context-free grammar G and parsing algorithm P, we say that G is a P grammar (e.g., an LL(1) grammar) if it can be parsed using that algorithm. By extension, for any context-free *language* L, we say that L is a P language if there exists a P grammar for L (this may not be the grammar we were given).

EXAMPLE 2.62
Separation of language classes

Containment relationships among the classes of languages accepted by the popular parsing algorithms appear in Figure C-2.38. Again, languages can be found in every region. Examples appear in Figure C-2.39; proofs are deferred to Exercise C-2.36.

It turns out that every context-free language that can be parsed deterministically has an SLR(1) grammar. In fact, any language that can parsed deterministically and in which no valid string can be extended to create another valid string (this is called the *prefix property*) has what is called an LR(0) grammar—one that can be parsed with no lookahead whatsoever! In the CFSM for such a grammar, any state containing an item with a $\bullet$ at the end will have no other item with a $\bullet$ in the middle. When such a state is reached, the parser can blindly reduce. If our scanner appends an explicit `$$` marker at end-of-file, it is easy to see that our (augmented) language will have the prefix property, and an LR(0) grammar must exist. At the same time, LR(0) grammars tend to be large and unintuitive. Among other things, they must generally avoid any epsilon productions: if an item $A \longrightarrow \varepsilon\bullet$ shares a state with an item in which the dot precedes a terminal, we won't be able to tell whether to "recognize" ε without peeking ahead. Moreover, for any given grammar, LR(0) parsers have no space or time advantage over SLR(1) or LALR(1). As a result, LR(0) tends not to be used in practice.

The relationships among language classes are not as rich as the relationships among grammar classes. Most real programming languages can be parsed by any

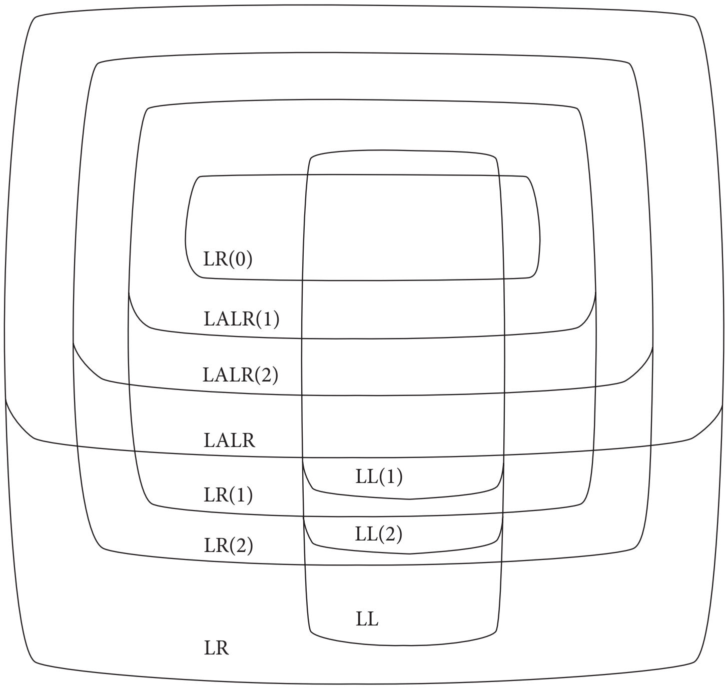


Figure 2.36 Containment relationships among popular grammar classes. Beyond the containments shown, SLL(k) is just inside LL(k), for $k \geq 2$; SLR(k) is just inside LALR(k), for $k \geq 1$.

LL(2) but not SLL:

$$\begin{aligned} S &\longrightarrow \texttt{a}\ A\ \texttt{a} \mid \texttt{b}\ A\ \texttt{b}\ \texttt{a} \\ A &\longrightarrow \texttt{b} \mid \varepsilon \end{aligned}$$

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

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

LR(0) but not LL:

$$\begin{aligned} S &\longrightarrow A\ \texttt{b} \\ A &\longrightarrow A\ \texttt{a} \mid \texttt{a} \end{aligned}$$

SLL(1) but not LALR:

$$\begin{aligned} S &\longrightarrow A\ \texttt{a} \mid B\ \texttt{b} \mid \texttt{c}\ C \\ C &\longrightarrow A\ \texttt{b} \mid B\ \texttt{a} \\ A &\longrightarrow D \\ B &\longrightarrow D \\ D &\longrightarrow \varepsilon \end{aligned}$$

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

$$\begin{aligned} S &\longrightarrow A\ \texttt{a}^{k-1}\ \texttt{b} \mid B\ \texttt{a}^{k-1}\ \texttt{c} \\ A &\longrightarrow \varepsilon \\ B &\longrightarrow \varepsilon \end{aligned}$$

LALR(1) but not SLR:

$$\begin{aligned} S &\longrightarrow \texttt{b}\ A\ \texttt{b} \mid A\ \texttt{c} \mid \texttt{a}\ \texttt{b} \\ A &\longrightarrow \texttt{a} \end{aligned}$$

LR(1) but not LALR:

$$\begin{aligned} S &\longrightarrow \texttt{a}\ C\ \texttt{a} \mid \texttt{b}\ C\ \texttt{b} \mid \texttt{a}\ D\ \texttt{b} \mid \texttt{b}\ D\ \texttt{a} \\ C &\longrightarrow \texttt{c} \\ D &\longrightarrow \texttt{c} \end{aligned}$$

Unambiguous but not LR:

$$S \longrightarrow \texttt{a}\ S\ \texttt{a} \mid \varepsilon$$

Figure 2.37 Examples of grammars in various regions of Figure C-2.36.

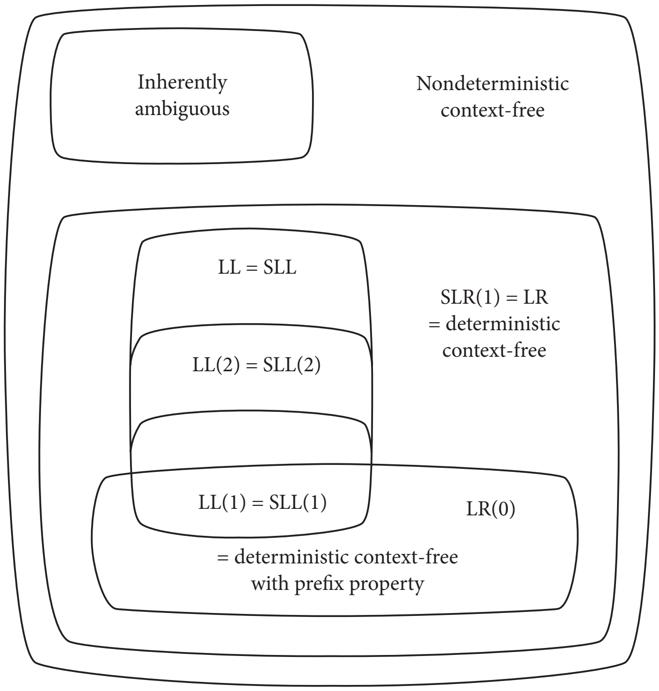

Inherently ambiguous
Nondeterministic context-free
LL = SLL
SLR(1) = LR = deterministic context-free
LL(2) = SLL(2)
LL(1) = SLL(1)
LR(0)
= deterministic context-free with prefix property

Figure 2.38 Containment relationships among popular language classes.

Nondeterministic language:
$\{\texttt{a}^n\texttt{b}^n\texttt{c} : n \geq 1\} \cup \{\texttt{a}^n\texttt{b}^{2n}\texttt{d} : n \geq 1\}$
Inherently ambiguous language:
$\{\texttt{a}^i\texttt{b}^j\texttt{c}^k : i = j \text{ or } j = k\ ;\ i, j, k \geq 1\}$
Language with LL(k) grammar but no LL(k−1) grammar:
$\{\texttt{a}^n(\texttt{b} \mid \texttt{c} \mid \texttt{b}^k\texttt{d})^n : n \geq 1\}$
Language with LR(0) grammar but no LL grammar:
$\{\texttt{a}^n\texttt{b}^n : n \geq 1\} \cup \{\texttt{a}^n\texttt{c}^n : n \geq 1\}$

Figure 2.39 Examples of languages in various regions of Figure C-2.38.

of the popular parsing algorithms, though the grammars are not always pretty, and special-purpose "hacks" may sometimes be required when a language is almost, but not quite, in a given class. The principal advantage of the more powerful parsing algorithms (e.g., full LR) is that they can parse a wider variety of grammars for a given language. In practice this flexibility makes it easier for the compiler writer to find a grammar that is intuitive and readable, and that facilitates the creation of semantic action routines.

✓ CHECK YOUR UNDERSTANDING

55. What formal machine captures the behavior of a scanner? A parser?

56. State three ways in which a real scanner differs from the formal machine.

57. What are the formal components of a DFA?

58. Outline the algorithm used to construct a regular expression equivalent to a given DFA.

59. What is the inherent "big-O" complexity of parsing with a simulated NPDA? Why is this worse than the $O(n^3)$ time mentioned in Section 2.3?

60. How many states are there in an LL(1) PDA? An SLR(1) PDA? Explain.

61. What are the *viable prefixes* of a CFG?

62. Summarize the proof that a DFA cannot recognize arbitrarily nested constructs.

63. Explain the difference between LL and SLL parsing.

64. Is every LL(1) grammar also LR(1)? Is it LALR(1)?

65. Does every LR language have an SLR(1) grammar?

66. Why are there never any epsilon productions in an LR(0) grammar?

67. Why are the containment relationships among grammar classes more complex than those among language classes?

2 Programming Language Syntax

2.6 Exercises

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

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

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

(a) with global FOLLOW sets, as in Example C-2.45.

(b) with local FOLLOW sets, as in Example C-2.47.

(c) with avoidance of "starter symbol" deletion, as in Example C-2.48.

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

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

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

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

2 Programming Language Syntax

2.7 Explorations

2.46 Experiment with syntax errors in your favorite compiler. Feed the compiler deliberate errors and comment on the quality of the recovery or repair. How often does it do the "right thing"? How often does it generate cascading errors? Speculate as to what sort of recovery or repair algorithm it might be using.

2.47 Spelling mistakes (typos in keywords and identifiers) are a common source of syntax and static semantic errors. Identifying such errors—and guessing what the user meant to type—could result in significantly better error recovery. Discuss how you might go about incorporating spelling correction into some existing error recovery system. (Hint: You might want to consult Morgan's early paper on this subject [Mor70].)

3 Names, Scopes, and Bindings

3.4 Implementing Scope

For both static and dynamic scoping, a language implementation must keep track of the name-to-object bindings in effect at each point in the program. The principal difference is *time*: with static scope the compiler uses a *symbol table* to track bindings at compile time; with dynamic scoping the interpreter or run-time system uses an *association list* or *central reference table* to track bindings at run time.

3.4.1 Symbol Tables

In a language with static scoping, the compiler uses an insert operation to place a name-to-object binding into the symbol table for each newly encountered declaration. When it encounters the use of a name that should already have been declared, the compiler uses a lookup operation to search for an existing binding. It is tempting to try to accommodate the visibility rules of static scoping by performing a remove operation to delete a name from the symbol table at the end of its scope. Unfortunately, several factors make this straightforward approach impractical:

- The ability of inner declarations to hide outer ones in most languages with nested scopes means that the symbol table has to be able to contain an arbitrary number of mappings for a given name. The lookup operation must return the innermost mapping, and outer mappings must become visible again at end of scope.
- Records (structures) and classes have some of the properties of scopes, but do not share their nicely nested structure. When it sees a record declaration, the semantic analyzer must remember the names of the record's fields (recursively, if records are nested). At the end of the declaration, the field names must become invisible. Later, however, whenever a variable of the record type appears in the program text (as in my_rec.field_name), the record fields must suddenly become visible again for the part of the reference after the dot. In object-oriented languages, member (field and method) names must become visible throughout

the methods of the class, even if (as in C++) the code for the methods can appear outside the class declaration.

- As noted in Section 3.3.3, names are sometimes used before they are declared. Algol and C, for example, permit *forward references* to labels. Pascal permits forward references in pointer declarations. Most object-oriented languages permit forward references to class members. Modula-3 permits forward references of all kinds.
- As noted in Section 3.3.3, C, C++, and Ada distinguish between the declaration of an object and its definition. Pascal has a similar mechanism for mutually recursive subroutines. When it sees a declaration, the compiler must remember any nonvisible details so that it can check the eventual definition for consistency. This operation is similar to remembering the field names of records and classes.
- While it may be desirable to forget names at the end of their scope, and even to reclaim the space they occupy in the symbol table, information about them may need to be saved for use by a *symbolic debugger* (Section 16.3.2). A debugger allows the user to manipulate a running program: starting it, stopping it, and reading and writing its data. In order to parse high-level commands, the debugger must have access to the compiler's symbol table, which the compiler typically saves in a hidden portion of the final machine-language program.

EXAMPLE 3.45
The LeBlanc-Cook symbol table

To accommodate these concerns, most compilers never delete anything from the symbol table. Instead, they manage visibility using enter_scope and leave_scope operations. Implementations vary from compiler to compiler; the approach described here is due to LeBlanc and Cook [CL83].

Each scope, as it is encountered, is assigned a serial number. The outermost scope (the one that contains the predefined identifiers) is given number 0. The scope containing programmer-declared global names is given number 1. Additional scopes are given successive numbers as they are encountered. All serial numbers are distinct; they do not represent the level of lexical nesting, except in as much as nested subroutines naturally end up with numbers higher than those of surrounding scopes. If language rules specify that a declaration should be visible only in the remainder of the current code block (not the preceding portion), we can even allocate a serial number for each such declaration, to capture the scope that is the remainder of the block.

All names, regardless of scope, are entered into a single large hash table, keyed by name. Each entry in the table then contains the symbol name, its category (variable, constant, type, procedure, field name, parameter, etc.), scope number, type (a pointer to another symbol table entry), and additional, category-specific fields.

In addition to the hash table, the symbol table has a *scope stack* that indicates, in order, the scopes that compose the current referencing environment. As the semantic analyzer scans the program, it pushes and pops this stack whenever it enters or leaves a scope, respectively. Entries in the scope stack contain the scope number, an indication of whether the scope is closed, and in some cases further information.

```
procedure lookup(name)
    pervasive := best := null
    apply hash function to name to find appropriate chain
    foreach entry e on chain
        if e.name = name        -- not something else with same hash value
            if e.scope = 0
                pervasive := e
            else
                foreach scope s on scope stack, top first
                    if s.scope = e.scope
                        best := e       -- closer instance
                        exit inner loop
                    elsif best ≠ null and then s.scope = best.scope
                        exit inner loop     -- won't find better
                    if s.closed
                        exit inner loop     -- can't see farther
    if best ≠ null
        while best is an import or export entry
            best := best.real_entry
        return best
    elsif pervasive ≠ null
        return pervasive
    else
        return null     -- name not found
```

Figure 3.17 LeBlanc-Cook symbol table lookup operation.

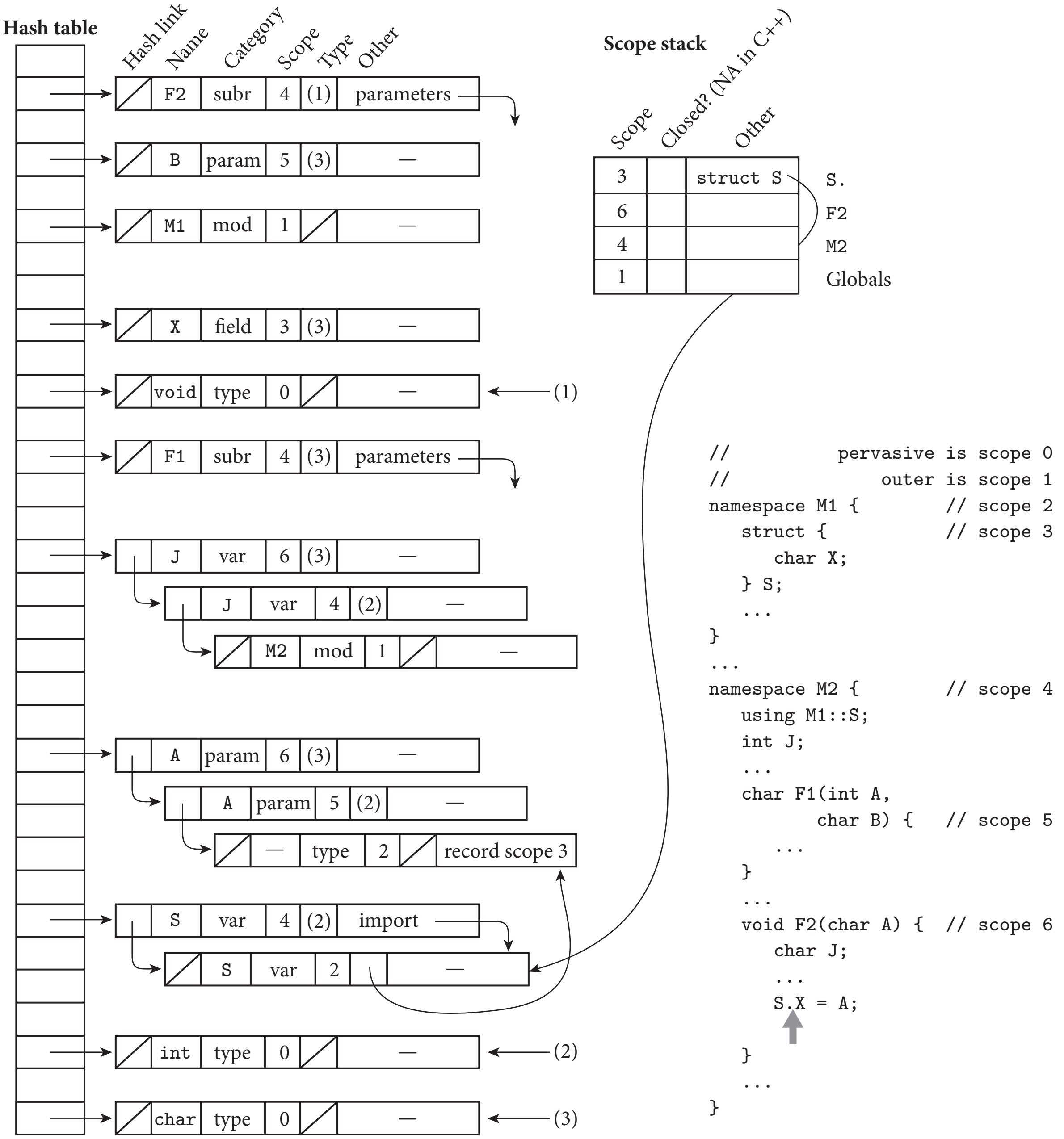
To look up a name in the table, we scan down the appropriate hash chain looking for entries that match the name we are trying to find. For each matching entry, we scan down the scope stack to see if the scope of that entry is visible. We look no deeper in the stack than the top-most closed scope. Imports and exports are made visible outside their normal scope by creating additional entries in the table; these extra entries contain pointers to the real entries. We don't have to examine the scope stack at all for entries with scope number 0: they are pervasive. Pseudocode for the lookup algorithm appears in Figure C-3.17.

EXAMPLE 3.46

Symbol table for a sample program

The lower right portion of Figure C-3.18 contains the skeleton of a C++ program. The remainder of the figure shows the configuration of the symbol table for the referencing environment of the grey arrow shown in function F2. At this point in the code, the scope stack contains four entries, representing, respectively, the (anonymous) type of structure S, function F2, namespace (module) M2, and the global scope. The scope for the anonymous type indicates the specific variable (i.e., S) to which names (fields) in this scope belong. The outermost, pervasive scope is not explicitly represented.

All of the entries for a given name appear on the same hash chain, since the table is keyed on name. In this example, we assume that hash collisions have placed M2 on the same chain as the Js, and the anonymous structure type (which will



Scope	Closed? (NA in C++)	Other	
3		struct S	S.
6			F2
4			M2
1			Globals

```
//          pervasive is scope 0
//              outer is scope 1
namespace M1 {        // scope 2
   struct {           // scope 3
      char X;
   } S;
   ...
}
...
namespace M2 {        // scope 4
   using M1::S;
   int J;
   ...
   char F1(int A,
          char B) {   // scope 5
      ...
   }
   ...
   void F2(char A) {  // scope 6
      char J;
      ...
      S.X = A;

   }
   ...
}
```

Figure 3.18 **LeBlanc-Cook symbol table for an example program in a language like C++.** The scope stack represents the referencing environment at the grey arrow shown in function F2. For the sake of clarity, the many pointers from type fields to the symbol table entries for `void`, `int`, and `char` are shown as parenthesized (1)s, (2)s, and (3)s, rather than as arrows.

have some arbitrary internal name) on the same chain as the As. Variable S has an extra entry, to make it directly visible inside M2, as requested by the using statement. When we are inside F2, a lookup operation on J will find F2's J; the J in M2 will be hidden by virtue of F2 being above M2 on the scope stack. The entry for the anonymous struct type indicates the scope number to be pushed onto the scope stack while resolving references to fields within objects of that type. The entry for each function contains the head pointer of a list that links together the subroutine's parameters, for use in analyzing calls (additional links of these chains are not shown). During code generation, many symbol table entries would contain additional fields, for such information as size and run-time address.

The second column of the scope stack is intended to indicate closed scopes. While C++ doesn't have any of these, we can imagine how they would work. For example, if M2 were closed, then names in the global scope, which appears below M2 in the scope stack, would not be visible at the indicated point in the code. Anything imported into M2 *would* be visible, because it would have an extra entry (like that of S) within M2's own scope.[1] If our language had exports (again, C++ does not), we would create extra entries in the *outer* scope, analogous to the ones we create in inner scopes for imports.

Classes, which we did not use in Figure C-3.18, would be handled much like a combination of namespaces and structures. Field and method names of the class would be visible to the class's methods, much as objects in a namespace are visible to all the namespace's code. Moreover, the entry for the class—like that of a structure type—would indicate the scope to be pushed onto the scope stack when the compiler has parsed a dot (.) or arrow (->) token and expects the next token to name a field or method of the class.

It is tempting to suggest extending a LeBlanc-Cook style symbol table to handle the visibility specifications common in object-oriented languages (e.g., the public, private, protected keywords of C++, to which we will return in Section 10.2.2), but this is probably a mistake. For one thing, such an extension would likely be quite messy. It is easy to make all the names in a scope visible, by pushing that scope onto the scope stack. It is also relatively easy to make a small number of names visible, by creating extra entries in other scopes, as we did for imports and exports. An intermediate option does not immediately present itself. More significantly, when the programmer attempts to use a field or method inappropriately, we probably want to generate an error message along the lines of "method m is private in class foo" instead of just "method name foo not found." This observation suggests employing a traditional lookup mechanism for class members, followed by a separate check for visibility in the current context. We consider this possibility in Exercise C-3.27.

[1] Recall that the using statement in C++ isn't an import in the usual sense: it just gives a simpler (unqualified) name to an already-visible object.

3.4.2 Association Lists and Central Reference Tables

Pictorial representations of the two principal implementations of dynamic scoping appear in Figures C-3.19 and C-3.20. Association lists (*A-lists*) are simple and elegant, but can be very inefficient. Central reference tables resemble a simplified LeBlanc-Cook symbol table, without the separate scope stack; they require more work at scope entry and exit than do association lists, but they make lookup operations fast.

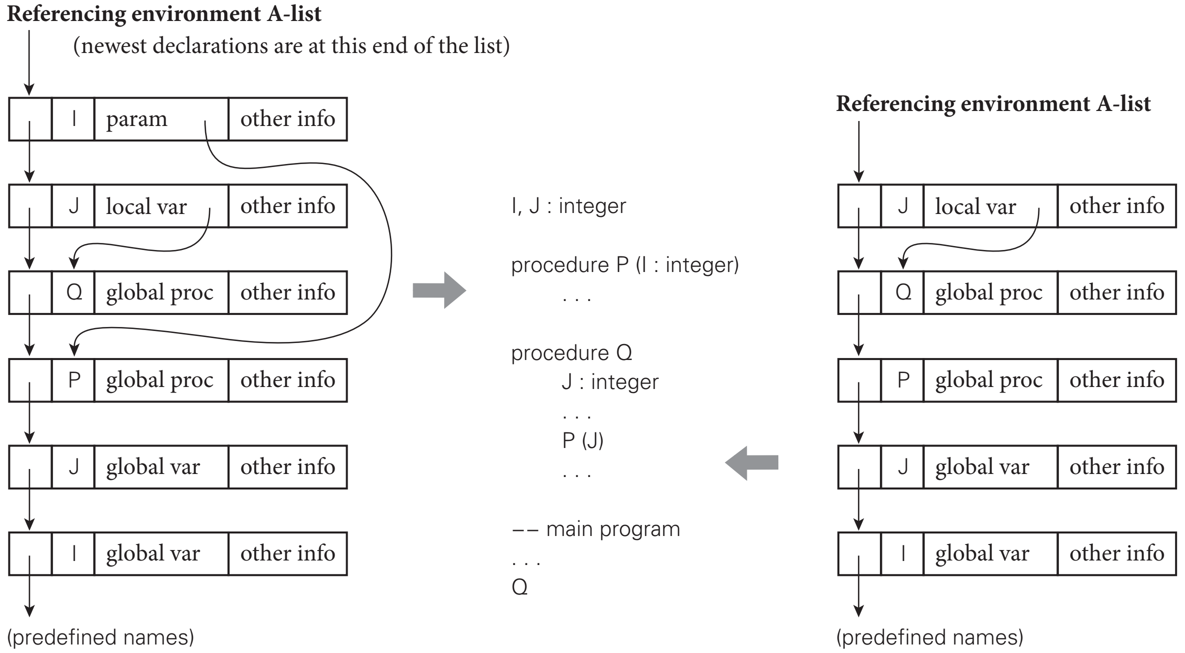
EXAMPLE 3.47
A-list lookup in Lisp

A-lists are widely used for dictionary abstractions in Lisp; they are supported by a rich set of built-in functions in most Lisp dialects. It is therefore natural for a simple Lisp interpreter to use an A-list to keep track of name-value bindings, and even to make this list explicitly visible to the running program. Since bindings are created when entering a scope, and destroyed when leaving or returning from a scope, the A-list functions as a stack. When execution enters a scope at run time, the interpreter pushes bindings for names declared in that scope onto the top of the A-list. When execution finally leaves a scope, these bindings are removed. To look up the meaning of a name in an expression, the interpreter searches from the top of the list until it finds an appropriate binding (or reaches the end of the list, in which case an error has occurred). Each entry in the list contains whatever information is needed to perform semantic checks (e.g., type checking, which we will consider in Section 7.2) and to find variables and other objects that occupy memory locations. In the left half of Figure C-3.19, the first (top) entry on the A-list represents the most recently encountered declaration: the I in procedure P. The second entry represents the J in procedure Q. Below these are the global symbols, Q, P, J, and I, and (not shown explicitly) any predefined names provided by the Lisp interpreter. ■

EXAMPLE 3.48
Central reference table

The problem with using an association list to represent a program's referencing environment is that it can take a long time to find a particular entry in the list, particularly if it represents an object declared in a scope encountered early in the program's execution, and now buried deep in the list. A central reference table is designed for faster access. It has one slot for every distinct name in the program. The table slot in turn contains a list (stack) of declarations encountered at run time, with the most recent occurrence at the beginning of the list. Looking up a name is now easy: the current meaning is found at the beginning of the list in the appropriate slot in the table. In the upper part of Figure C-3.20, the first entry on the I list is the I in procedure P; the second is the global I. If the program is compiled and the set of names is known at compile time, then each name can have a statically assigned slot in the table, which the compiled code can refer to directly. If the program is not compiled, or the set of names is not statically known, then a hash function will need to be used at run time to find the appropriate slot. ■

When control enters a new scope at run time, entries must be pushed onto the beginning of every list in the central reference table whose name is (re)declared in that scope. When control leaves a scope for the final time, these entries must be popped. The work involved is somewhat more expensive than pushing and popping an A-list, but not dramatically more so, and lookup operations are now



```
I, J : integer

procedure P (I : integer)
        . . .

procedure Q
        J : integer
        . . .
        P (J)
        . . .

-- main program
. . .
Q
```

Figure 3.19 Dynamic scoping with an association list. The left side of the figure shows the referencing environment at the point in the code indicated by the adjacent grey arrow: after the main program calls Q and it in turn calls P. When searching for I, one will find the parameter at the beginning of the A-list. The right side of the figure shows the environment at the other grey arrow: after P returns to Q. When searching for I, one will find the global definition.

much faster. In contrast to the symbol table of a compiler for a language with static scoping, central reference table entries for a given scope do not need to be saved when the scope completes execution; the space can be reclaimed.

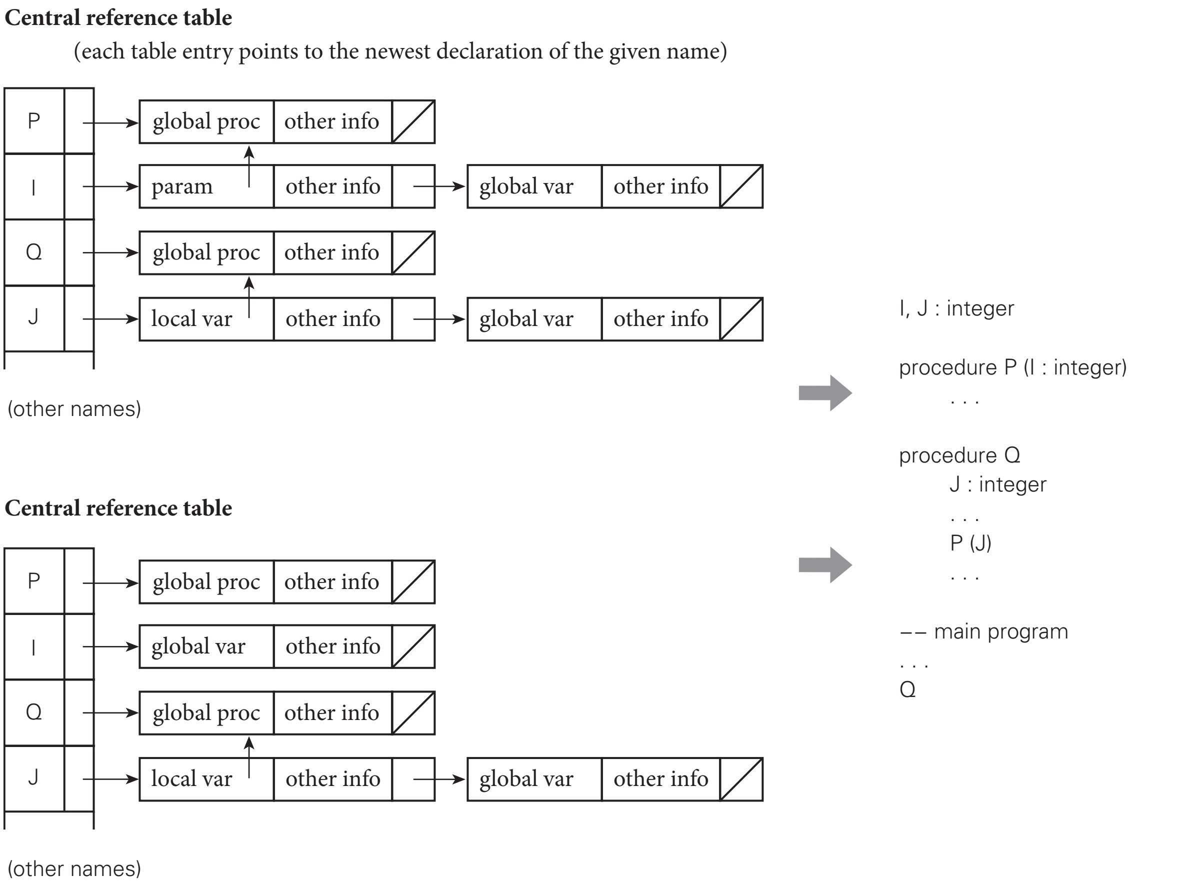
Within the Lisp community, implementation of dynamic scoping via an association list is sometimes called *deep binding*, because the lookup operation may need to look arbitrarily deep in the list. Implementation via a central reference table is sometimes called *shallow binding*, because it finds the current association at the head of a given reference chain. Unfortunately, the terms "deep and shallow binding" are also more widely used for a completely different purpose, discussed in Section 3.6. To avoid potential confusion, some authors use "deep and shallow access" [Seb19] or "deep and shallow search" [Fin96] for the implementations of dynamic scoping.

Closures with Dynamic Scoping

(This subsection is best read after Section 3.6.1.)

EXAMPLE 3.49
A-list closures

If an association list is used to represent the referencing environment of a program with dynamic scoping, the referencing environment in a closure can be represented by a top-of-stack (beginning of A-list) pointer (Figure C-3.21). When



```
I, J : integer

procedure P (I : integer)
      . . .

procedure Q
      J : integer
      . . .
      P (J)
      . . .

–– main program
. . .
Q
```

Figure 3.20 **Dynamic scoping with a central reference table.** The upper half of the figure shows the referencing environment at the point in the code indicated by the upper grey arrow: after the main program calls Q and it in turn calls P. When searching for I, one will find the parameter at the beginning of the chain in the I slot of the table. The lower half of the figure shows the environment at the lower grey arrow: after P returns to Q. When searching for I, one will find the global definition.

a subroutine is called through a closure, the main pointer to the referencing environment A-list is temporarily replaced by the pointer from the closure, making any bindings created since the closure was created (P's I and J in the figure) temporarily invisible. New bindings created *within* the subroutine (or in any subroutine it calls) are pushed using the temporary pointer. Because the A-list is represented by pointers (rather than an array), the effect is to have two lists—one representing the caller's referencing environment and the other the temporary referencing environment resulting from use of the closure—that share their older entries. When Q returns to P in our example, the original head of the A-list will be restored, making P's I and J visible again. ■

```
procedure P(procedure C)
        declare I, J
        call C

procedure F
        declare I

procedure Q
        declare J
        call F

–– main program
        call P(Q)
```

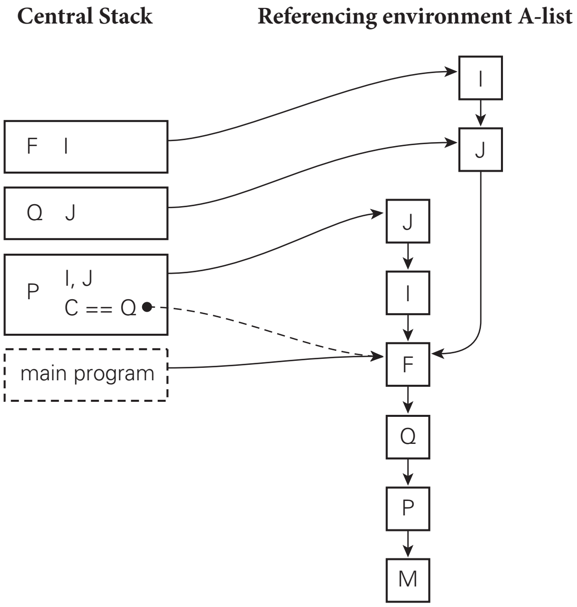


Figure 3.21 **Capturing the A-list in a closure.** Each frame in the stack has a pointer to the current beginning of the A-list, which the run-time system uses to look up names. When the main program passes Q to P with deep binding, it bundles its A-list pointer in Q's closure (dashed arrow). When P calls C (which is Q), it restores the bundled pointer. When Q elaborates its declaration of J (and F elaborates its declaration of I), the A-list is temporarily bifurcated.

With a central reference table implementation of dynamic scoping, the creation of a closure is more complicated. In the general case, it may be necessary to copy the entire main array of the central table and the first entry on each of its lists. Space and time overhead may be reduced if the compiler or interpreter is able to determine that only some of the program's names will be used by the subroutine in the closure (or by things that the subroutine may call). In this case, the environment can be saved by copying the first entries of the lists for only the names that will be used. When the subroutine is called through the closure, these entries can then be pushed onto the beginnings of the appropriate lists in the central reference table. Additional code must be executed to remove them again after the subroutine returns.

✓ CHECK YOUR UNDERSTANDING

43. List the basic operations provided by a symbol table.

44. Outline the implementation of a LeBlanc-Cook style symbol table.

45. Why don't compilers generally remove names from the symbol table at the ends of their scopes?

46. Describe the *association list* (*A-list*) and *central reference table* data structures used to implement dynamic scoping. Summarize the tradeoffs between them.

47. Explain how to implement deep binding by capturing the referencing environment A-list in a closure. Why are closures harder to build with a central reference table?

3 Names, Scopes, and Bindings

3.8 Separate Compilation

Probably the most straightforward mechanisms for separate compilation can be found in module-based languages such as Modula-2, Modula-3, and Ada, which allow a module to be divided into a declaration part (or *header*) and an implementation part (or *body*). As we noted in Section 3.3.4, the header contains all and only the information needed by users of the module (or needed by the compiler in order to compile such a user); the body contains the rest.

As a matter of software engineering practice, a design team will typically define module interfaces early in the lifetime of a project, and codify these interfaces in the form of module headers. Individual team members or subteams will then work to implement the module bodies. While doing so, they can compile their code successfully using the headers for the other modules. Using preliminary copies of the bodies, they may also be able to undertake a certain amount of testing.

In a simple implementation, only the body of a module needs to be compiled into runnable code: the compiler can read the header of module *M* when compiling the body of *M*, and also when compiling the body of any module that uses *M*. In a more sophisticated implementation, the compiler can avoid the overhead of repeatedly scanning, parsing, and analyzing *M*'s header by translating it into a symbol table, which is then accessed directly when compiling the bodies of *M* and its users. Most Ada implementations compile their module headers. Implementations of Modula-2 and 3 vary: some work one way, some the other.

As a practical matter, many languages allow the header of a module to be subdivided into a "public" part, which specifies the interface to the rest of the program, and a "private" part, which is not visible outside the module, but is needed by the compiler, for example to determine the storage requirements of opaque types. Ideally, one would include in the header of a module only that information that the programmer needs to know to use the abstraction(s) that the module provides. Restricted exports, and the public and private portions of headers, are compromises introduced to allow the compiler to generate code in the face of separate compilation.

At some point prior to execution, modules that have been separately compiled must be "glued together" to form a single program. This job is the task of the *linker*. At the very least, the linker must resolve cross-module references (loads, stores, jumps) and *relocate* any instructions whose encoding depends on the location of certain modules in the final program. Typically it also checks to make sure that users and implementors of a given interface agree on the version of the header file used to define that interface. In some environments, the linker may perform additional tasks as well, including certain kinds of interprocedural (whole-program) code improvement. We will return to the subject of linking in Chapters 15 and 16.

3.8.1 Separate Compilation in C

The initial version of C was designed at Bell Laboratories around 1970. It has evolved considerably over the years, but not, for the most part, in the area of separate compilation. Here the language remains comparatively primitive. In particular, there is in general no way for the compiler or the linker to detect inconsistencies among declarations or uses of a name in different files. The C89 standards committee introduced a new explanation of separate compilation based on the notion of *linkage*, but this served mainly to clarify semantics, not to change them. The current rules can be summarized as follows (certain details and special cases are omitted):

- If the declaration of a global object (variable or function) contains the word `static`, then the object has *internal linkage*, and is identified with (linked to) any other internally linked declaration of the same name in the same file.
- If the declaration of a function does not contain the keyword `static`, then it has *external linkage*, and is identified with any other (nonstatic) declaration of the same function in any file of the program. (A function declaration may consist of just the header.)
- If the declaration of a variable contains the keyword `extern`, then the variable has the same linkage as any visible, internally or externally linked declaration of the same name appearing earlier in the file. If there is no earlier declaration, then the variable has external linkage, and is identified with any other declaration of the same external variable in any file of the program. In other words, files in the same program that contain matching external variable declarations actually share the same variable. A global variable also has external linkage if its declaration says neither `static` nor `extern`.
- If an object is declared with both internal and external linkage, the behavior of the program is undefined.
- An object (variable or function) that is externally linked must have a *definition* in exactly one file of a program. A variable is defined when it is given an initial

value, or is declared at the global level without the `extern` keyword. A function is defined when its body (code) is given.

Many C implementations prior to C89 relaxed the final rule to permit zero or one definitions of an external variable; some permitted more than one. In these implementations, the linker unified multiple definitions, and created an implicit definition for any variable (or set of linked variables) for which the program contained only declarations.

The "linkage" rules of C89 provide a way to associate names in one file with names in another file. The rules are most easily understood in terms of their implementation. Most language-independent linkers are designed to deal with *symbols*: character-string names for locations in a machine-language program. The linker's job is to assign every symbol a location in the final program, and to embed the address of the symbol in every machine-language instruction that makes a reference to it. To do this job, the linker needs to know which symbols can be used to resolve unbound references in other files, and which are local to a given file. C89 rules suffice to provide this information. For the programmer, however, there is no formal notion of *interface*, and no mechanism to make a name visible in some, but not all files. Moreover, nothing ensures that the declarations of an external object found in different files will be compatible: it is entirely possible, for example, to declare an external variable as a multifield record in one file and as a floating-point number in another. The compiler is not required to catch such errors, and the resulting bugs can be very difficult to find.

Header Files

Fortunately, C programmers have developed conventions on the use of external declarations that tend to minimize errors in practice. These conventions rely on the *file inclusion* facility of a macro preprocessor. The programmer creates files in pairs that correspond roughly to the interface and the implementation of a module. The name of an interface file ends with `.h`; the name of the corresponding implementation file ends with `.c`. Every object *defined* in the `.c` file is *declared* in the `.h` file. At the beginning of the `.c` file, the programmer inserts a directive that is treated as a special form of comment by the compiler, but that causes the preprocessor to include a verbatim copy of the corresponding `.h` file. This inclusion operation has the effect of placing "forward" declarations of all the module's objects at the beginning of its implementation file. Any inconsistencies with definitions later in the file will result in error messages from the compiler. The programmer also instructs the preprocessor at the top of each `.c` file to include a copy of the `.h` files for all of the modules on which the `.c` file depends. As long as the preprocessor includes identical copies of a given `.h` file in all the `.c` files that use its module, no inconsistent declarations will occur. Unfortunately, it is easy to forget to recompile one or more `.c` files when a `.h` file is changed, and this can lead to very subtle bugs. Tools like Unix's `make` utility help minimize such errors by keeping track of the dependences among modules.

Namespaces

Even with the convention of header files, C89 still suffers from the lack of scoping beyond the level of an individual file. In particular, all global names must be distinct, across all files of a program, and all libraries to which it links. Some coding standards encourage programmers to embed a module's name in the name of each of its external objects (e.g., `scanner_nextSym`), but this practice can be awkward, and is far from universal.

To address this limitation, C++ introduced a `namespace` mechanism that generalizes the scoping already provided for classes and functions, breaks the tie between module and compilation unit, and strengthens the interface conventions of `.h` files. Any collection of names can be declared inside a `namespace`:

EXAMPLE 3.50
Namespaces in C++

```
namespace foo {
    class foo_type_1;           // declaration
    ...
}
```

Actual definitions of the objects within `foo` can then appear in any file:

```
class foo::foo_type_1 { ...     // full definition
```

Definitions of objects declared in different namespaces can appear in the same file if desired. ■

EXAMPLE 3.51
Using names from another namespace

A C++ programmer can access the objects in a namespace using *fully qualified* names, or by *importing* (`using`) them explicitly:

```
foo::foo_type_1 my_first_obj;
```

or

```
using foo::foo_type_1;
...
foo_type_1 my_first_obj;
```

or

```
using namespace foo;    // import everything from foo
...
foo_type_1 my_first_obj;
```

There is no notion of export; all objects with external linkage in a namespace are visible elsewhere if imported or accessed with their qualified name. Note that linkage remains the foundation for separate compilation: `.h` files are merely a convention. ■

3.8.2 Packages and Automatic Header Inference

EXAMPLE 3.52
Packages in Java

The separate compilation facilities of Java and C# eliminate .h files. Java introduces a formal notion of module, called a *package*. Every *compilation unit*, which may be a file or (in some implementations) a record in a database, belongs to exactly one package, but a package may consist of many compilation units, each of which begins with an indication of the package to which it belongs:

```
package foo;
public class FooType1 { ...
```

Unless explicitly declared as public, a class in Java is visible in all and only those compilation units that belong to the same package. ■

EXAMPLE 3.53
Using names from another package

As in C++, a compilation unit that needs to use classes from another package can access them using fully qualified names, or via name-at-a-time or package-at-a-time import:

```
foo.FooType1 myFirstObj;
```

or

```
import foo.FooType1;
...
FooType1 myFirstObj;
```

or

```
import foo.*;           // import everything from foo
...
FooType1 myFirstObj;
```

■

When asked to import names from package *M*, the Java compiler will search for *M* in a standard (but implementation-dependent) set of places, and will recompile it if appropriate (i.e., if only source code is found, or if the target code is out of date). The compiler will then *automatically* extract the information that would have been needed in a C++ .h file or an Ada or Modula-3 header. If the compilation of *M* requires other packages, the compiler will search for them as well, recursively.

C# follows Java's lead in extracting header information automatically from complete class definitions. Its module-level syntax, however, is based on the namespaces of C++, which allow a single file to contain fragments of multiple namespaces. There is also no notion of standard search path in C#: to build a complete program, the programmer must provide the compiler with a complete list of all the files required.

To mimic the software engineering practice of early header file construction, a Java or C# design team can create skeleton versions of (the public classes of) its packages or namespaces, which can then be used, concurrently and independently, by the programmers responsible for the full versions.

3.8.3 Module Hierarchies

In Modula and Ada, the programmer can create a hierarchy of modules within a single compilation unit by means of lexical nesting (module C, for example, may be declared inside of module B, which in turn is declared inside of module A). In a similar vein, the Ada 95, Java, or C# programmer can create a hierarchy of separately compiled modules by means of *multipart names:*

EXAMPLE 3.54
Multipart package names

```
package A.B is ...          -- Ada 95

package A.B; ...            // Java

namespace A.B { ...         // C#
```

In these examples package A.B is said to be a *child* of package A. In Ada 95 and C# the child behaves as though it had been nested inside of the parent, so that all the names in the parent are automatically visible. In Java, by contrast, multipart names work by convention only: there is no special relationship between packages A and A.B. If A.B needs to refer to names in A, then A must make them public, and A.B must import them. Child packages in Ada 95 are reminiscent of derived classes in C++, except that they support a module-as-manager style of abstraction, rather than a module-as-type style (more on this in sidebar 10.3). ■

✓ **CHECK YOUR UNDERSTANDING**

48. What purpose(s) does separate compilation serve?
49. What does it mean for an external variable to be *linked* in C?
50. Summarize the C conventions for use of .h and .c files.
51. Describe the difference between a compilation unit and a C++ or C# *namespace*.
52. Explain why Ada and similar languages separate the header of a module from its body. Explain how Java and C# get by without.

DESIGN & IMPLEMENTATION

3.12 Separate compilation

The evolution of separate compilation mechanisms from early C and Fortran, through C++, Modula-3, Ada, and finally Java and C#, reflects a move from an implementation-centric viewpoint to a more programmer-centric viewpoint. Interestingly, the ability to have zero definitions of an externally linked variable in certain early implementations of C is inherited from Fortran: the assembly language mnemonic corresponding to a declaration without a definition is .common (for a mechanism known as common blocks in Fortran).

3 Names, Scopes, and Bindings

3.10 Exercises

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

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

(a) function `F1` in Figure 3.4.

(b) procedure `set_seed` in Figure 3.7.

3.26 In Example C-3.45 we suggested that the implementor of a language in which declarations are visible only in the remainder of the current code block might choose to introduce a new nested scope for every declaration. This would, of course, lead to a very deep scope stack. If that turned out to be a performance problem for the compiler, explain how you might layer a caching mechanism on top of the standard lookup algorithm to eliminate most of the slow-down.

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

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

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

(b) Exercise 3.14.

3.29 Repeat the previous exercise for a central reference table.

3.30 Consider the following tiny program in C:

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

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

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

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

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

3.31 Consider the following file from some larger C program:

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

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

static int b;
extern int c;
```

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

3.32 Modula-2 provides no way to divide the header of a module into a public part and a private part: everything in the header is visible to the users of the module. Is this a major shortcoming? Are there disadvantages to the public/private division (e.g., as in Ada)? (For hints, see Section 10.2.)

3 Names, Scopes, and Bindings

3.11 Explorations

3.44 Using your favorite compiler, generate assembly language for some simple programs with debugger support enabled (on a Unix system, this will probably require the `-g` and `-S` command-line switches). Look through the result for debugger information. Can you decipher any of it? You may want to look ahead to Section 16.3.2, and to consult a manual for your system's object file format (on a modern Unix system, look for documentation on DWARF).

3.45 Learn about the *reflection* mechanisms of Java, C#, Prolog, Perl, PHP, Tcl, Python, or Ruby, all of which allow a program to inspect and reason about its own symbol table at run time. How complete are these mechanisms? (For example, can a program inspect symbols that aren't currently in scope?) What is reflection good for? What uses should be considered good or bad programming practice? For more ideas, see Section 16.3.1.

3.46 Learn about the `typeglob` mechanism of Perl, which allows a program to modify its own symbol table at run time. What are `typeglobs` good for? (See Sidebar 14.9 for some initial pointers.)

3.47 Create a C program in which a variable is exported from one file and imported by another, but the declarations in the files disagree with respect to type. You should be able to arrange for the program to compile and link successfully, but behave incorrectly. Try the same thing in Ada or C++. What happens?

3.48 Investigate the use of module hierarchies in the standard libraries of C++, Java, and C#. How is each organized? How fine grain is the division into separate headers or packages? Can you suggest an explanation for any major differences you find?

4 Program Semantics

4.6 Attribute Grammars

In this section we examine *attribute grammars*, an alternate formalism for describing and implementing the semantics of a programming language. Intuitively, we can think of attribute grammars as a generalization of action routines in which the compiler designer no longer needs to specify exactly when to execute each routine—and in which the execution need not necessarily be interleaved with parsing. Alternatively, we can think of attribute grammars as a more imperative alternative to inference rules: instead of providing rules that indicate what can be inferred about the meaning of nodes in a syntax tree, we provide code to compute the values of attributes (fields) of tree nodes, either in a syntax tree or in the original parse tree.

EXAMPLE 4.24

Bottom-up CFG for constant expressions

As a starting point, in a parse tree context, consider an LR (bottom-up) grammar for arithmetic expressions composed of constants with precedence and associativity, adapted from Example 2.8:[1]

$$
\begin{aligned}
E &\longrightarrow E \; \texttt{+} \; T\\
E &\longrightarrow E \; \texttt{-} \; T\\
E &\longrightarrow T\\
T &\longrightarrow T \; \texttt{*} \; F\\
T &\longrightarrow T \; \texttt{/} \; F\\
T &\longrightarrow F\\
F &\longrightarrow \texttt{-} \; F\\
F &\longrightarrow \texttt{(} \; E \; \texttt{)}\\
F &\longrightarrow \texttt{const}
\end{aligned}
$$

[1] The addition of semantic rules tends to make attribute grammars quite a bit more verbose than context-free grammars. For the sake of brevity, many of the examples in this section use very short symbol names: E instead of *expr*, TT instead of *term_tail*.

1. $E_1 \longrightarrow E_2$ `+` T	▷ $\mathsf{E_1.val := sum(E_2.val, T.val)}$
2. $E_1 \longrightarrow E_2$ `-` T	▷ $\mathsf{E_1.val := difference(E_2.val, T.val)}$
3. $E \longrightarrow T$	▷ E.val := T.val
4. $T_1 \longrightarrow T_2$ `*` F	▷ $\mathsf{T_1.val := product(T_2.val, F.val)}$
5. $T_1 \longrightarrow T_2$ `/` F	▷ $\mathsf{T_1.val := quotient(T_2.val, F.val)}$
6. $T \longrightarrow F$	▷ T.val := F.val
7. $F_1 \longrightarrow$ `-` F_2	▷ $\mathsf{F_1.val := additive\_inverse(F_2.val)}$
8. $F \longrightarrow$ `(` E `)`	▷ F.val := E.val
9. $F \longrightarrow$ `const`	▷ F.val := const.val

Figure 4.16 **A simple attribute grammar for constant expressions, using the standard arithmetic operations.** Each semantic rule is introduced by a ▷ sign.

This grammar will generate all properly formed constant expressions over the basic arithmetic operators, but it says nothing about their meaning. To tie these expressions to mathematical concepts (as opposed to, say, floor tile patterns or dance steps), we could use inference rules, as discussed in the main text, but we

EXAMPLE 4.25
Bottom-up AG for constant expressions

can also use *attributes*. In our expression grammar, we associate a val attribute with each E, T, F, and `const` in the grammar. The intent is that for any symbol S, S.val will be the meaning, as an arithmetic value, of the token string derived from S. We assume that the val of a `const` is provided to us by the scanner. We must then invent a set of rules for each production, to specify how the vals of different symbols are related. The resulting *attribute grammar* (AG) is shown in Figure C-4.16.

In this simple grammar, every production has a single rule. We shall see more complicated grammars later, in which productions can have several rules. The rules come in two forms. Those in productions 3, 6, 8, and 9 are known as *copy rules*; they specify that one attribute should be a copy of another. The other rules invoke *semantic functions* (sum, quotient, additive_inverse, etc.). In this example, the semantic functions are all familiar arithmetic operations. In general, they can be arbitrarily complex functions specified by the language designer. Each semantic function takes an arbitrary number of arguments (each of which must be an attribute of a symbol in the current production—no global variables are allowed), and each computes a single result, which must likewise be assigned into an attribute of a symbol in the current production. When more than one symbol of a production has the same name, subscripts are used to distinguish them. These subscripts are solely for the benefit of the semantic functions; they are not part of the context-free grammar itself. ■

In a strict definition of attribute grammars, copy rules and semantic function calls are the only two kinds of permissible rules. In our examples we use a ▷ symbol to introduce each code fragment corresponding to a single rule. In practice, it is common to allow rules to consist of small fragments of code in some well-defined notation (e.g., the language in which a compiler is being written), so that simple semantic functions can be written out "in-line." In this relaxed notation, the rule for the first production in Figure C-4.16 might be simply E_1.val := E_2.val + T.val.

EXAMPLE 4.26

Top-down AG to count the elements of a list

As another example, suppose we wanted to count the elements of a comma-separated list:

$$\begin{array}{lll} L \longrightarrow \texttt{id}\ LT & & \triangleright\ \text{L.c := 1 + LT.c} \\ LT \longrightarrow\ \texttt{,}\ L & & \triangleright\ \text{LT.c := L.c} \\ LT \longrightarrow \varepsilon & & \triangleright\ \text{LT.c := 0} \end{array}$$

Here the rule on the first production sets the c (count) attribute of the left-hand side to one more than the count of the tail of the right-hand side. Like explicit semantic functions, in-line rules are not allowed to refer to any variables or attributes outside the current production. We will relax this restriction when we relate attribute grammars to action routines in Subsection C-4.6.2. ■

Neither the notation for semantic functions (whether in-line or explicit) nor the types of the attributes themselves is intrinsic to the notion of an attribute grammar. The purpose of the grammar is simply to associate meaning with the nodes of a parse tree or syntax tree. Toward that end, we can use any notation and types whose meanings are already well defined. In Examples C-4.25 and C-4.26, we associated numeric values with the symbols in a CFG—and thus with parse tree nodes—using semantic functions drawn from ordinary arithmetic. In a compiler or interpreter for a full programming language, the attributes of tree nodes might include

- for an identifier, a reference to information about it in the symbol table
- for an expression, its type
- for a statement or expression, a reference to corresponding code in the compiler's intermediate form
- for almost any construct, an indication of the file name, line, and column where the corresponding source code begins
- for any internal node, a list of semantic errors found in the subtree below

For purposes other than translation—for example, in a theorem prover or machine-independent language definition—attributes might be drawn from the disciplines of denotational, operational, or axiomatic semantics. Operational semantics were discussed in Section 4.3; interested readers can find references to other alternatives in the Bibliographic Notes at the end of the chapter.

4.6.1 Evaluating Attributes

EXAMPLE 4.27

Decoration of a parse tree

The process of evaluating attributes is called *annotation* or *decoration* of the parse tree (it also applies to syntax trees, as we shall see in Section C-4.6.3). Figure C-4.17 shows how to decorate the parse tree for the expression `(1 + 3) * 2`, using the AG of Figure C-4.16. Once decoration is complete, the value of the overall expression can be found in the val attribute of the root of the tree. ■

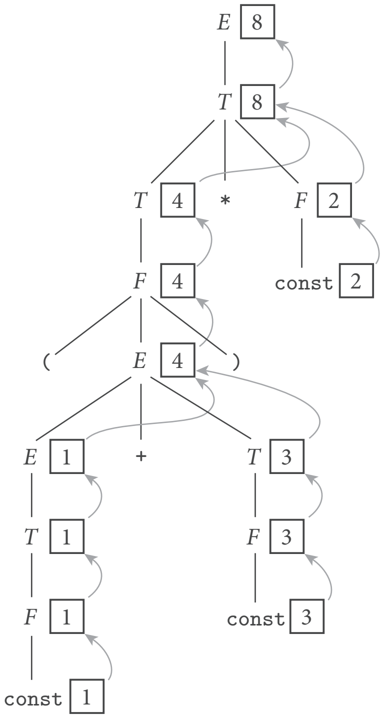


Figure 4.17 Decoration of a parse tree for `(1 + 3) * 2`, using the attribute grammar of Figure C-4.16. The val attributes of symbols are shown in boxes. Curving arrows show the attribute flow, which is strictly upward in this case. Each box holds the output of a single semantic rule; the arrow(s) entering the box indicate the input(s) to the rule. At the second level of the tree, for example, the two arrows pointing into the box with the 8 represent application of the rule T_1.val := product(T_2.val, F.val).

Synthesized Attributes

The attribute grammar of Figure C-4.16 is very simple. Each symbol has at most one attribute (the punctuation marks have none). Moreover, they are all so-called *synthesized attributes*: their values are calculated (synthesized) only in productions in which their symbol appears on the left-hand side. For annotated parse trees like the one in Figure C-4.17, this means that the *attribute flow*—the pattern in which information moves from node to node—is entirely bottom-up.

An attribute grammar in which all attributes are synthesized is said to be *S-attributed*. The arguments to semantic functions in an S-attributed grammar are always attributes of symbols on the right-hand side of the current production, and the return value is always placed into an attribute of the left-hand side of the production. Tokens (terminals) often have intrinsic properties (e.g., the character-string representation of an identifier or the value of a numeric constant); in a compiler these are synthesized attributes initialized by the scanner.

Inherited Attributes

When we considered the construction of syntax trees during top-down parsing (Example 4.6 and Figure 4.6), we found that we needed to place action routines *within* the right-hand sides of productions, so that the left operands of an arithmetic operator could be passed into the subtree that would contain the right operand. In a similar vein—and for similar reasons—we will encounter situations in which attribute values will need to be calculated when their symbol is on the right-hand side of the current production. Such attributes are said to be *inherited*. They allow contextual information to flow into a symbol from above or from the side, so that the rules of that production can be enforced in different ways (or generate different values) depending on surrounding context. Symbol table information is commonly passed from symbol to symbol by means of inherited attributes. Inherited attributes of the root of the parse tree can also be used to represent the external environment (characteristics of the target machine, command-line arguments to the compiler, etc.).

EXAMPLE 4.28

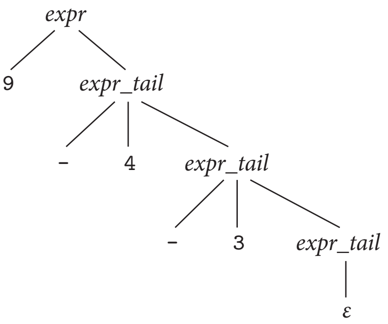
Top-down CFG and parse tree for subtraction

As a simple example of inherited attributes, consider the following fragment of an LL(1) expression grammar (here covering only subtraction):

expr ⟶ `const` *expr_tail*

expr_tail ⟶ `-` `const` *expr_tail* | ε

For the expression `9 - 4 - 3`, we obtain the following parse tree:

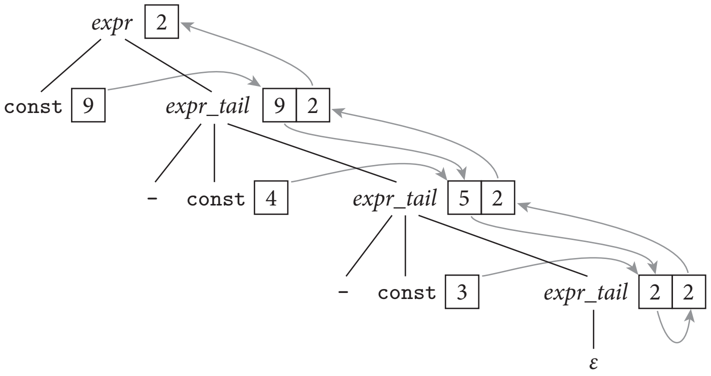


If we want to create an attribute grammar that accumulates the value of the overall expression into the root of the tree, we have a problem: because subtraction is left associative, we cannot summarize the right subtree of the root with a single numeric value. If we want to decorate the tree bottom-up, with an S-attributed grammar, we must be prepared to describe an arbitrary number of right operands in the attributes of the top-most *expr_tail* node (see Exercise C-4.23). This is indeed possible, but it defeats the purpose of the formalism: in effect, it requires us to embed the entire tree into the attributes of a single node, and do all the real work inside a single semantic function at the root.

EXAMPLE 4.29

Decoration with left-to-right attribute flow

If, however, we are allowed to pass attribute values not only bottom-up but also left-to-right in the tree, then we can pass the `9` into the top-most *expr_tail* node, where it can be combined (in proper left-associative fashion) with the `4`. The resulting 5 can then be passed into the middle *expr_tail* node, combined with the 3 to make 2, and then passed upward to the root:



EXAMPLE 4.30
Top-down AG for subtraction

To effect this style of decoration, we need the following attribute rules:

expr ⟶ `const` *expr_tail*
▷ expr_tail.st := const.val
▷ expr.val := expr_tail.val

$expr\_tail_1$ ⟶ `- const` $expr\_tail_2$
▷ $\text{expr\_tail}_2\text{.st} := \text{expr\_tail}_1\text{.st} - \text{const.val}$
▷ $\text{expr\_tail}_1\text{.val} := \text{expr\_tail}_2\text{.val}$

expr_tail ⟶ ε
▷ expr_tail.val := expr_tail.st

In each of the first two productions, the first rule serves to copy the left context (value of the expression so far) into a "subtotal" (st) attribute; the second rule copies the final value from the right-most leaf back up to the root. In the *expr_tail* nodes of the picture in Example C-4.29, the left box holds the st attribute; the right holds val.

EXAMPLE 4.31
Top-down AG for constant expressions

We can flesh out the grammar fragment of Example C-4.28 to produce a more complete expression grammar, as shown (with shorter symbol names) in Figure C-4.18. The underlying CFG for this grammar accepts the same language as the one in Figure C-4.16, but where that one was SLR(1), this one is LL(1). Attribute flow for a parse of `(1 + 3) * 2`, using the LL(1) grammar, appears in Figure C-4.19. As in the grammar fragment of Example C-4.30, the value of the left operand of each operator is carried into the *TT* and *FT* productions by the st (subtotal) attribute. The relative complexity of the attribute flow arises from the fact that operators are left associative, but the grammar cannot be left recursive: the left and right operands of a given operator are thus found in separate productions. Grammars to perform semantic analysis for practical languages generally require some non-S-attributed flow.

1. $E \longrightarrow T\ TT$
   ▷ TT.st := T.val ▷ E.val := TT.val

2. $TT_1 \longrightarrow$ `+` $T\ TT_2$
   ▷ TT$_2$.st := TT$_1$.st + T.val ▷ TT$_1$.val := TT$_2$.val

3. $TT_1 \longrightarrow$ `-` $T\ TT_2$
   ▷ TT$_2$.st := TT$_1$.st − T.val ▷ TT$_1$.val := TT$_2$.val

4. $TT \longrightarrow \varepsilon$
   ▷ TT.val := TT.st

5. $T \longrightarrow F\ FT$
   ▷ FT.st := F.val ▷ T.val := FT.val

6. $FT_1 \longrightarrow$ `*` $F\ FT_2$
   ▷ FT$_2$.st := FT$_1$.st × F.val ▷ FT$_1$.val := FT$_2$.val

7. $FT_1 \longrightarrow$ `/` $F\ FT_2$
   ▷ FT$_2$.st := FT$_1$.st ÷ F.val ▷ FT$_1$.val := FT$_2$.val

8. $FT \longrightarrow \varepsilon$
   ▷ FT.val := FT.st

9. $F_1 \longrightarrow$ `-` F_2
   ▷ F$_1$.val := − F$_2$.val

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

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

Figure 4.18 **An attribute grammar for constant expressions based on an LL(1) CFG.** In this grammar several productions have two semantic rules.

Attribute Flow

Just as a context-free grammar does not specify how it should be parsed, an attribute grammar does not specify the order in which attribute rules should be invoked. Put another way, both notations are *declarative*: they define a set of valid parse trees, but they don't say how to build or decorate them. Among other things, this means that the order in which attribute rules are listed for a given production is immaterial; attribute flow may require them to execute in any order. If, in Figure C-4.18, we were to reverse the order in which the rules appear in productions 1, 2, 3, 5, 6, and/or 7 (listing the rule for symbol.val first), it would be a purely cosmetic change; the grammar would not be altered.

We say an attribute grammar is *well defined* if its rules determine a unique set of values for the attributes of every possible parse tree. An attribute grammar is *noncircular* if it never leads to a parse tree in which there are cycles in the attribute flow graph—that is, if no attribute, in any parse tree, ever depends (transitively) on itself. (A grammar can be circular and still be well defined if attributes are guaranteed to converge to a unique value.) As a general rule, practical attribute grammars tend to be noncircular.

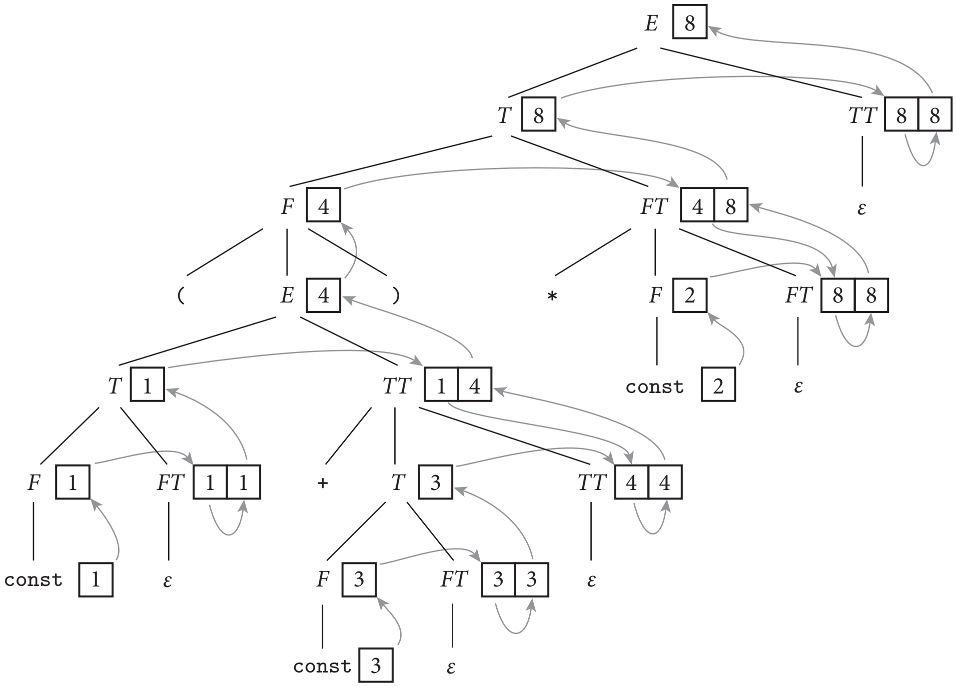


Figure 4.19 Decoration of a top-down parse tree for `(1 + 3) * 2`, using the AG of Figure C-4.18. Curving arrows again indicate attribute flow; the arrow(s) entering a given box represent the application of a single semantic rule. Flow in this case is no longer strictly bottom-up, but it is still left-to-right. At *FT* and *TT* nodes, the left box holds the `st` attribute; the right holds `val`.

An algorithm that decorates parse trees by invoking the rules of an attribute grammar in an order consistent with the tree's attribute flow is called a *translation scheme*. Perhaps the simplest scheme is one that makes repeated passes over a tree, invoking any semantic function whose arguments have all been defined, and stopping when it completes a pass in which no values change. Such a scheme is said to be *oblivious*, in the sense that it exploits no special knowledge of either the parse tree or the grammar. It will halt only if the grammar is well defined. Better performance, at least for noncircular grammars, may be achieved by a *dynamic* scheme that tailors the evaluation order to the structure of a given parse tree—for example, by constructing a topological sort of the attribute flow graph and then invoking rules in an order consistent with the sort.

The fastest translation schemes, however, tend to be *static*—based on an analysis of the structure of the attribute grammar itself, and then applied mechanically to any tree arising from the grammar. Like LL and LR parsers, linear-time static translation schemes can be devised only for certain restricted classes of grammars. S-attributed grammars, such as the one in Figure C-4.16, form the simplest such class. Because attribute flow in an S-attributed grammar is strictly bottom-up, attributes can be evaluated by visiting the nodes of the parse tree in exactly the same order that those nodes are generated by an LR-family parser. In fact, the attributes can be evaluated on the fly during a bottom-up parse, thereby interleaving parsing and semantic analysis (attribute evaluation).

The attribute grammar of Figure C-4.18 is a good bit messier than that of Figure C-4.16, but it is still *L-attributed*: its attributes can be evaluated by visiting the nodes of the parse tree in a single left-to-right, depth-first traversal (the same order in which they are visited during a top-down parse—see Figure C-4.19). If we say that an attribute A.s *depends on* an attribute B.t if B.t is ever passed to a semantic function that returns a value for A.s, then we can define L-attributed grammars more formally with the following two rules: (1) each synthesized attribute of a left-hand-side symbol depends only on that symbol's own inherited attributes or on attributes (synthesized or inherited) of the production's right-hand-side symbols, and (2) each inherited attribute of a right-hand-side symbol depends only on inherited attributes of the left-hand-side symbol or on attributes (synthesized or inherited) of symbols to its left in the right-hand side.

Because L-attributed grammars permit rules that initialize attributes of the left-hand side of a production using attributes of symbols on the right-hand side, every S-attributed grammar is also an L-attributed grammar. The reverse is not the case: S-attributed grammars do not permit the initialization of attributes on the right-hand side, so there are L-attributed grammars that are not S-attributed.

S-attributed attribute grammars are the most general class of attribute grammars for which evaluation can be implemented on the fly during an LR parse. L-attributed grammars are the most general class for which evaluation can be implemented on the fly during an LL parse. If we interleave semantic analysis (and possibly intermediate code generation) with parsing, then a bottom-up parser must in general be paired with an S-attributed translation scheme; a top-down parser must be paired with an L-attributed translation scheme. (Depending on the structure of the grammar, it is often possible for a bottom-up parser to accommodate some non-S-attributed attribute flow; we consider this possibility in Section C-4.6.4.) If we choose to separate parsing and semantic analysis into separate passes, then the code that builds the parse tree or syntax tree must still use an S-attributed or L-attributed translation scheme (as appropriate), but the semantic analyzer can use a more powerful scheme if desired. There are certain tasks that are easiest to accomplish with a non-L-attributed scheme. Examples include the generation of code for "short-circuit" Boolean expressions (to be discussed in Sections 6.1.5 and 6.4.1) and the type checking of mutually recursive functions (Section 3.3.3).

Building a Syntax Tree

EXAMPLE 4.32

Bottom-up and top-down AGs to build a syntax tree

If we choose not to interleave parsing and semantic analysis, we still need to add attribute rules to the context-free grammar, but they serve only to create the syntax tree—not to enforce semantic rules or generate code. Figures C-4.20 and C-4.21 contain bottom-up and top-down attribute grammars, respectively, to build a syntax tree for constant expressions. The attributes in these grammars hold neither numeric values nor target code fragments; instead they point to nodes

$E_1 \longrightarrow E_2$ `+` T
▷ E_1.ptr := bin_op(E_2.ptr, "+", T.ptr)

$E_1 \longrightarrow E_2$ `-` T
▷ E_1.ptr := bin_op(E_2.ptr, "−", T.ptr)

$E \longrightarrow T$
▷ E.ptr := T.ptr

$T_1 \longrightarrow T_2$ `*` F
▷ T_1.ptr := bin_op(T_2.ptr, "×", F.ptr)

$T_1 \longrightarrow T_2$ `/` F
▷ T_1.ptr := bin_op(T_2.ptr, "÷", F.ptr)

$T \longrightarrow F$
▷ T.ptr := F.ptr

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

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

$F \longrightarrow$ `const`
▷ F.ptr := int_lit(const.val)

Figure 4.20 Bottom-up (S-attributed) attribute grammar to construct a syntax tree. The symbol $^+/_-$ is used (as it is on calculators) to indicate change of sign.

of the syntax tree. Function int_lit returns a pointer to a newly allocated syntax tree node containing the value of a constant. Functions un_op and bin_op return pointers to newly allocated syntax tree nodes containing a unary or binary operator, respectively, and pointers to the supplied operand(s). Bottom-up and top-down construction of syntax trees for `(1 + 3) * 2` is analogous to that of Figures 4.5 and 4.8, respectively, in the main text.

4.6.2 Action Routines and Attribute Grammars

The astute reader will have noticed the similarity between Figures 4.4 and C-4.20, and between Figures 4.6 and C-4.21. Indeed, the action routines we introduced in Section 4.2 are simply an implementation, provided by most parser-generator tools, of attribute grammars with a manually specified static translation scheme. Each action routine is a semantic function that the programmer (grammar writer) instructs the compiler to execute at a particular point in the parse.

One difference between the action routines of Figures 4.4 and 4.6 and the semantic functions of attribute grammars is that the former just return a value, while the former can set multiple attributes. While some tools (e.g., the University of Minnesota's attribute grammar-based Silver system) allow an action routine to modify multiple attributes, many popular parser generators, including `yacc/bison`

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

$TT_1 \longrightarrow$ `+` $T\ TT_2$
    ▷ TT$_2$.st := bin_op(TT$_1$.st, "+", T.ptr)
    ▷ TT$_1$.ptr := TT$_2$.ptr

$TT_1 \longrightarrow$ `-` $T\ TT_2$
    ▷ TT$_2$.st := bin_op(TT$_1$.st, "−", T.ptr)
    ▷ TT$_1$.ptr := TT$_2$.ptr

$TT \longrightarrow \varepsilon$
    ▷ TT.ptr := TT.st

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

$FT_1 \longrightarrow$ `*` $F\ FT_2$
    ▷ FT$_2$.st := bin_op(FT$_1$.st, "×", F.ptr)
    ▷ FT$_1$.ptr := FT$_2$.ptr

$FT_1 \longrightarrow$ `/` $F\ FT_2$
    ▷ FT$_2$.st := bin_op(FT$_1$.st, "÷", F.ptr)
    ▷ FT$_1$.ptr := FT$_2$.ptr

$FT \longrightarrow \varepsilon$
    ▷ FT.ptr := FT.st

$F_1 \longrightarrow$ `-` F_2
    ▷ F$_1$.ptr := un_op("+/_", F$_2$.ptr)

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

$F \longrightarrow$ `const`
    ▷ F.ptr := int_lit(const.val)

Figure 4.21 **Top-down (L-attributed) attribute grammar to construct a syntax tree.** Here the st attribute, like the ptr attribute (and unlike the st attribute of Figure C-4.18), is a pointer to a syntax tree node.

and JavaCC, provide only the simpler return-value mechanism. In these tools, an action routine that needs to modify more than one attribute can return a record with a separate field for each.

✓ **CHECK YOUR UNDERSTANDING**

39. What is an *attribute grammar*?

40. What is the difference between *synthesized* and *inherited* attributes?

41. Give two examples of information that is typically passed through inherited attributes.

42. What is *attribute flow*?
43. What does it mean for an attribute grammar to be *S-attributed*? *L-attributed*? *Noncircular*? What is the significance of these grammar classes?
44. What is the difference between a semantic function and an action routine?

4.6.3 Semantic Analysis with Attribute Grammars

In our discussion so far we have used attribute grammars solely to decorate parse trees. Attribute grammars can also be used, however, to decorate syntax trees. To define semantic analyses over syntax trees using attribute grammars, we can simply attach semantic rules to the productions of an abstract grammar. These rules define the attribute flow of a syntax tree in exactly the same way that semantic rules attached to the productions of a context-free grammar are used to define the attribute flow of a parse tree. We will use an abstract grammar in the remainder of this section to perform static semantic checking. Additional semantic rules could be used to generate intermediate code.

EXAMPLE 4.33
Abstract AG for the calculator language with types

A complete abstract attribute grammar for our calculator language with types can be constructed using the node classes, variants, and attributes shown in Figure C-4.22. The grammar itself appears in Figure C-4.23. Once decorated, the *program* node at the root of the syntax tree will contain a list, in a synthesized attribute, of all static semantic errors in the program. (The list will be empty if the program is free of such errors.) Each *stmt* or *expr* node has an inherited attribute symtab that contains a list, with types, of all identifiers declared to the left in the tree. Each *stmt* node also has an inherited attribute errors_in that lists all static semantic errors found to its left in the tree, and a synthesized attribute errors_out to propagate the final error list back to the root. Each *expr* node has one synthesized attribute that indicates its type and another that contains a list of any static semantic errors found inside. To avoid cascading messages when an error is found early in

DESIGN & IMPLEMENTATION

4.6 Attribute evaluators

Automatic evaluators based on formal attribute grammars are popular in language research projects because they save developer time when the language definition changes. They are popular in syntax-based editors and incremental compilers because they save execution time: when a small change is made to a program, the evaluator may be able to "patch up" tree decorations significantly faster than it could rebuild them from scratch. For the typical compiler, however, semantic analysis based on a formal attribute grammar is overkill: it has higher overhead than action routines or ad-hoc traversal of a syntax tree, and doesn't really save the compiler writer that much work.

		Attributes	
Class of node	**Variants**	**Inherited**	**Synthesized**
program	—	—	location, errors
stmt	*int_decl, real_decl, assign, read, write, null*	symtab, errors_in	location, errors_out
expr	*int_lit, real_lit, var, bin_op, float, trunc*	symtab	location, type, errors name (*var* only)

Figure 4.22 Classes of nodes for the abstract attribute grammar of Figure C-4.23. All variants of a given class have all the class's attributes.

our pass over the syntax tree, we adopt the technique introduced in Section 4.4.2: we associate a pseudotype called error with any symbol table entry or expression for which we have already generated a message.

Though it takes a bit of checking to verify the fact, our attribute grammar is noncircular and well defined. No attribute is ever assigned a value more than once. (The helper routines at the end of Figure C-4.23 should be thought of as macros, rather than semantic functions. For the sake of brevity we have passed them entire tree nodes as arguments. Each macro calculates the values of two different attributes. Under a strict formulation of attribute grammars each macro would be replaced by two separate semantic functions, one per calculated attribute.)

EXAMPLE 4.34

Decorating a tree with the AG of the previous Example

Figure C-4.24 uses the grammar of Figure C-4.23 to decorate the syntax tree of Figure 4.2. The pattern of attribute flow appears considerably messier than in previous examples in this section, but this is simply because type checking is more complicated than calculating constants or building a syntax tree. Symbol table information flows along the chain of *stmt*s and down into *expr* trees. The *int_decl* and *real_decl* nodes add new information; other nodes simply pass the table along. Ideally, when an undeclared identifier is encountered, we would enter it into the symbol table with an "error" designation, to suppress further messages about the same identifier; we have not shown that code here.

Type information is synthesized at *var*, *assign*, and *expr* leaves by looking up an identifier's name in the symbol table. The information then propagates upward within an expression tree, and is used to type-check operators and assignments (the latter don't appear in this example). Error messages flow along the chain of *stmt*s via the errors_in attributes, and then back to the root via the errors_out attributes. Messages also flow up out of *expr* trees. Wherever a type check is performed, the type attribute may be used to help create a new message to be appended to the growing message list.

In our example grammar we accumulate error messages into a synthesized attribute of the root of the syntax tree. In an ad hoc attribute evaluator we might be tempted to print these messages on the fly as the errors are discovered. In practice, however, particularly in a multipass compiler, it makes sense to buffer the messages, so they can be interleaved with messages produced by other phases of the compiler, and printed in program order at the end of compilation.

```
$program \longrightarrow stmt$
    ▷ stmt.symtab := null
    ▷ program.errors := stmt.errors_out
    ▷ stmt.errors_in := null

$stmt_1 \longrightarrow$ int $id\ stmt_2$
    ▷ declare_name(id.name, stmt$_1$, stmt$_2$, int)
    ▷ stmt$_1$.errors_out := stmt$_2$.errors_out

$stmt_1 \longrightarrow$ real $id\ stmt_2$
    ▷ declare_name(id.name, stmt$_1$, stmt$_2$, real)
    ▷ stmt$_1$.errors_out := stmt$_2$.errors_out

$stmt_1 \longrightarrow$ read $id\ stmt_2$
    ▷ stmt$_2$.symtab := stmt$_1$.symtab
    ▷ if ⟨id.name, ?⟩ ∈ stmt$_1$.symtab
          stmt$_2$.errors_in := stmt$_1$.errors_in
      else
          stmt$_2$.errors_in := stmt$_1$.errors_in + [id.name "undefined at" id.location]
    ▷ stmt$_1$.errors_out := stmt$_2$.errors_out

$stmt_1 \longrightarrow$ write $expr\ stmt_2$
    ▷ expr.symtab := stmt$_1$.symtab
    ▷ stmt$_2$.symtab := stmt$_1$.symtab
    ▷ stmt$_2$.errors_in := stmt$_1$.errors_in + expr.errors
    ▷ stmt$_1$.errors_out := stmt$_2$.errors_out

$stmt_1 \longrightarrow id$ := $expr\ stmt_2$
    ▷ expr.symtab := stmt$_1$.symtab
    ▷ stmt$_2$.symtab := stmt$_1$.symtab
    ▷ if ⟨id.name, A⟩ ∈ stmt$_1$.symtab            –– for some type A
          if A $\neq$ error and expr.type $\neq$ error and A $\neq$ expr.type
              stmt$_2$.errors_in := stmt$_1$.errors_in + ["type clash at" :=.location]
          else
              stmt$_2$.errors_in := stmt$_1$.errors_in + expr.errors
      else
          stmt$_2$.errors_in := stmt$_1$.errors_in + [id.name "undefined at" id.location]
              + expr.errors
    ▷ stmt$_1$.errors_out := stmt$_2$.errors_out

$null : stmt \longrightarrow \varepsilon$
    ▷ stmt.errors_out := stmt.errors_in
```

Figure 4.23 Attribute grammar to decorate an abstract syntax tree for the calculator language with types. We use square brackets to delimit error messages and pointed brackets to delimit symbol table entries. Juxtaposition indicates concatenation within error messages; the '+' and '–' operators indicate insertion and removal in lists. We assume that every node has been initialized by the scanner or by action routines in the parser to contain an indication of the location (line and column) at which the corresponding construct appears in the source (see Exercise C-4.36). The '?' symbol is used as a "wild card"; it matches any type. *(continued)*

```
$expr \longrightarrow var$
    ▷ if ⟨var.name, A⟩ ∈ expr.symtab              -- for some type A
            expr.errors := null
            expr.type := A
      else
            expr.errors := [var.name ''undefined at'' var.location]
            expr.type := error

$expr \longrightarrow n$
    ▷ expr.type := int

$expr \longrightarrow r$
    ▷ expr.type := real

$expr_1 \longrightarrow expr_2\ op\ expr_3$
    ▷ $expr_2$.symtab := $expr_1$.symtab
    ▷ $expr_3$.symtab := $expr_1$.symtab
    ▷ check_types($expr_1$, $expr_2$, op, $expr_3$)

$expr_1 \longrightarrow$ float($expr_2$)
    ▷ $expr_2$.symtab := $expr_1$.symtab
    ▷ convert_type($expr_2$, $expr_1$, int, real, ''float of non-int'')

$expr_1 \longrightarrow$ trunc($expr_2$)
    ▷ $expr_2$.symtab := $expr_1$.symtab
    ▷ convert_type($expr_2$, $expr_1$, real, int, ''trunc of non-real'')
```

Figure 4.23 *(continued on next page)*

One could convert our attribute grammar into executable code using an automatic attribute evaluator generator. Alternatively, one could create an ad hoc evaluator in the form of mutually recursive subroutines (Exercise C-4.35). In the latter case attribute flow would be explicit in the calling sequence of the routines. We could then choose if desired to keep the symbol table in global variables, rather than passing it from node to node through attributes. Most compilers employ the ad hoc approach.

✓ **CHECK YOUR UNDERSTANDING**

45. What patterns of attribute flow can be captured easily with action routines?

46. Some compilers perform all semantic checks and intermediate code generation in action routines. Others use action routines to build a syntax tree and then perform semantic checks and intermediate code generation in separate traversals of the syntax tree. Discuss the tradeoffs between these two strategies.

47. What sort of information do action routines typically keep in global variables, rather than in attributes?

48. How can a semantic analyzer avoid the generation of cascading error messages?

```
macro declare_name(name, cur_stmt, next_stmt : syntax_tree_node; t : type)
    if ⟨name, ?⟩ ∈ cur_stmt.symtab
        next_stmt.errors_in := cur_stmt.errors_in + [“redefinition of” name “at” cur_stmt.location]
        next_stmt.symtab := cur_stmt.symtab − ⟨name, ?⟩ + ⟨name, error⟩
    else
        next_stmt.errors_in := cur_stmt.errors_in
        next_stmt.symtab := cur_stmt.symtab + ⟨name, t⟩

macro check_types(result, operand1, op, operand2)
    if operand1.type = error or operand2.type = error
        result.type := error
        result.errors := operand1.errors + operand2.errors
    else if operand1.type ≠ operand2.type
        result.type := error
        result.errors := operand1.errors + operand2.errors + [“type clash at” op.location]
    else
        result.type := operand1.type
        result.errors := operand1.errors + operand2.errors

macro convert_type(old_expr, new_expr : syntax_tree_node; from_t, to_t : type; msg : string)
    if old_expr.type = from_t or old_expr.type = error
        new_expr.errors := old_expr.errors
        new_expr.type := to_t
    else
        new_expr.errors := old_expr.errors + [msg “at” old_expr.location]
        new_expr.type := error
```

Figure 4.23 *(continued)*

4.6.4 Space Management for Attributes

Any attribute evaluation method requires space to hold the attributes of the grammar symbols. In an attribute grammar based on the abstract grammar of explicit syntax trees, the obvious approach is to store attributes in the nodes of the tree themselves. In a context-free grammar with action routines, the analogous approach applies only if we are building an explicit parse tree—and usually we're not. This means we need to find a way to keep track of the attributes of symbols we have seen (or predicted) but not yet finished parsing. The details differ in bottom-up and top-down parsers.

For a bottom-up parser with an S-attributed grammar, it is straightforward to maintain an *attribute stack* that directly mirrors the parse stack: next to every state number on the parse stack is an attribute record for the symbol we shifted when we entered that state. Entries in the attribute stack are pushed and popped automatically by the parser driver; space management is not an issue for the writer of action routines. Complications arise if we try to achieve the effect of inherited attributes, but these can be accommodated within the basic attribute-stack framework.

For a top-down parser with an L-attributed grammar, we have two principal options. The first option is automatic, but more complex than for bottom-up

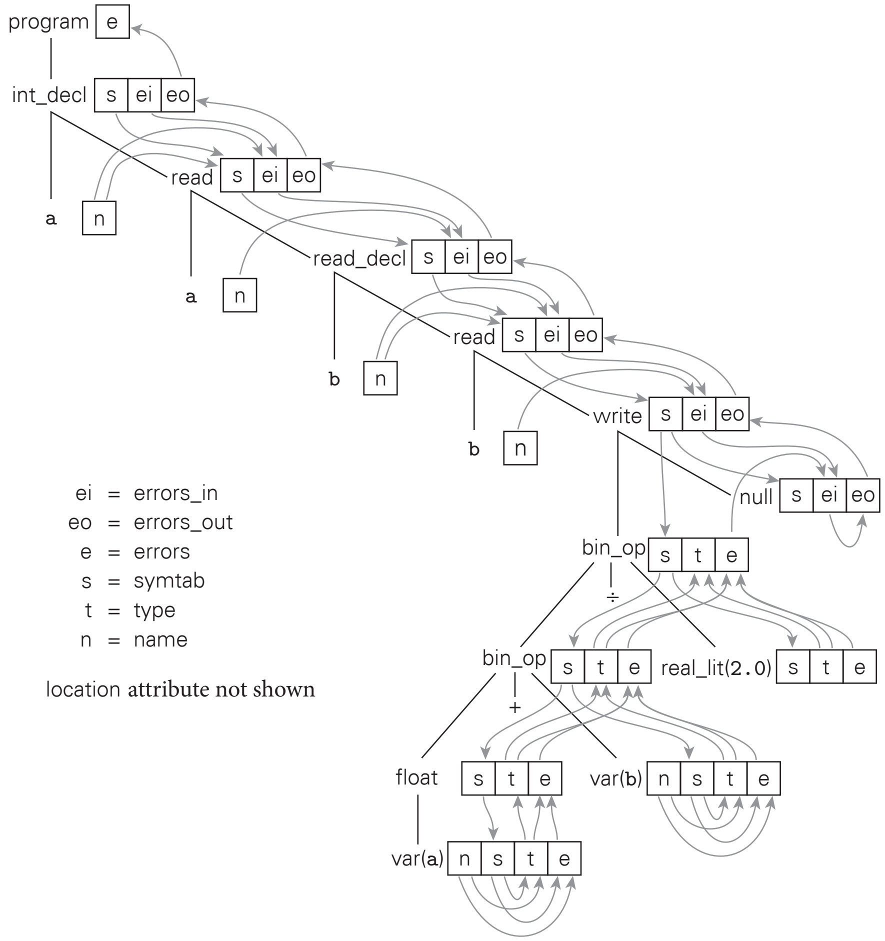


Figure 4.24 Decoration of the syntax tree of Figure 4.2, using the grammar of Figure C-4.23. Location information, which we assume has been initialized in every node by the parser, contributes to error messages, but does not otherwise propagate through the tree.

grammars. It still uses an attribute stack, but one that does not mirror the parse stack, because it must store information about symbols that have already been parsed. The second option has lower space overhead, and saves time by "short-cutting" copy rules, but requires action routines to allocate and deallocate space for attributes explicitly.

In both bottom-up and top-down parsers, it is common for some of the contextual information for action routines to be kept in global variables. The symbol table in particular is usually global. Rather than pass its full contents through attributes from one production to the next, we pass an indication of the currently active

1. (
2. (1
3. (F_1
4. (T_1
5. (E_1
6. (E_1 +
7. (E_1 + 3
8. (E_1 + F_3
9. (E_1 + T_3
10. (E_4
11. (E_4)
12. F_4
13. T_4
14. T_4 *
15. T_4 * 2
16. T_4 * F_2
17. T_8
18. E_8

Figure 4.25 Parse/attribute stack trace for `(1 + 3) * 2`, using the grammar of Figure C-4.16. Subscripts represent val attributes; they are not meant to distinguish among instances of a symbol.

scope. Lookups in the global table then use this scope information to obtain the right referencing environment.

In this subsection, we consider attribute space management in more detail. Using bottom-up and top-down grammars for arithmetic expressions, we illustrate automatic management for both bottom-up and top-down parsers, as well as the ad hoc option for top-down parsers.

Bottom-Up Evaluation

EXAMPLE 4.35
Stack trace for bottom-up parse, with action routines

Figure C-4.25 shows a trace of the parse and attribute stack for `(1 + 3) * 2`, using the attribute grammar of Figure C-4.16. For the sake of clarity, we show a single, combined stack for the parser and attribute evaluator, and we omit the CFSM state numbers.

It is easy to evaluate the attributes of symbols in this grammar, because the grammar is S-attributed. In an automatically generated parser, such as those produced by `yacc/bison`, the attribute rules associated with the productions of the grammar in Figure C-4.16 would constitute action routines, to be executed when their productions are recognized. For `yacc/bison`, they would be written in C, with "pseudostructs" to name the attribute records of the symbols in each production. Attributes of the left-hand side symbol would be accessed as fields of the pseudostruct `$$`. Attributes of right-hand side symbols would be accessed as fields of the pseudostructs `$1`, `$2`, etc. To get from line 9 to line 10, for example, in the trace of Figure C-4.25, we would use an action routine version of the first rule of the grammar in Figure C-4.16: $$.val = $1.val + $3.val. ■

When a bottom-up action routine is executed, the attribute records for symbols on the right-hand side of the production can be found in the top few entries of the attribute stack. The attribute record for the symbol on the left-hand side of the production (i.e., `$$`) will not yet lie in the stack: it is the task of the action routine to initialize this record. After the action routine completes, the parser pops the right-hand side records off the attribute stack and replaces them with `$$`. In `yacc`/`bison`, if no action routine is specified for a given production, the default action is to "copy" `$1` into `$$`. Since `$$` will occupy the same location, once pushed, that `$1` occupied before being popped, this "copy" can be effected without doing any work.

EXAMPLE 4.36
Finding inherited attributes in "buried" records

Inherited Attributes. Unfortunately, it is not always easy to write an S-attributed grammar. A simple example in which inherited attributes are desirable arises in C or Fortran-style variable declarations, in which a type name precedes the list of variable names:

dec ⟶ *type id_list*
id_list ⟶ `id`
id_list ⟶ *id_list* `,` `id`

Let us assume that *type* has a synthesized attribute `tp` that contains a pointer to the symbol table entry for the type in question. Ideally, we should like to pass this attribute into *id_list* as an inherited attribute, so that we may enter each newly declared identifier into the symbol table, complete with type indication, as it is encountered. When we recognize the production *id_list* ⟶ `id`, we know that the top record on the attribute stack will be the one for `id`. But we know more than this: the next record down must be the one for *type*. To find the type of the new entry to be placed in the symbol table, we may safely inspect this "buried" record. Though it does not belong to a symbol of the current production, we can count on its presence because there is no other way to reach the *id_list* ⟶ `id` production.

Now what about the `id` in *id_list* ⟶ *id_list* `,` `id`? This time the top three records on the attribute stack will be for the right-hand symbols `id`, `,`, and *id_list*. Immediately below them, however, we can still count on finding the entry for *type*, waiting for the *id_list* to be completed so that *dec* can be recognized. Using nonpositive indices for pseudostructs below the current production, we can write action routines as follows:

dec ⟶ *type id_list*
id_list ⟶ `id` { declare_id ($1.name, $0.tp) }
id_list ⟶ *id_list* `,` `id` { declare_id ($3.name, $0.tp) }

Records deeper in the attribute stack could be accessed as $–1, $–2, and so on. While *id_list* appears in two places in this grammar fragment, both occurrences are guaranteed to lie above a *type* record in the attribute stack, the first because it lies next to *type* in a right-hand side, and the second by induction, because it is the beginning of the yield of the first.

EXAMPLE 4.37

Grammar fragment requiring context

Unfortunately, there are grammars in which a symbol that needs inherited attributes occurs in productions in which the underlying symbols are not the same. We can still handle inherited attributes in such cases, but only by modifying the underlying context-free grammar. An example can be found in languages like Perl, in which the meaning of an expression (and of the identifiers and operators within it) depends on the *context* in which that expression appears. Some Perl contexts expect arrays. Others expect numbers, strings, or Booleans. To correctly analyze an expression, we must pass the expectations of the context into the expression subtree as inherited attributes. Here is a grammar fragment that captures the problem:

$$\begin{array}{lll}
\mathit{stmt} & \longrightarrow & \texttt{id := } \mathit{expr} \\
 & \longrightarrow & \ldots \\
 & \longrightarrow & \texttt{if } \mathit{expr} \texttt{ then } \mathit{stmt} \\
\mathit{expr} & \longrightarrow & \ldots
\end{array}$$

Within the production for *expr*, the parser doesn't know whether the surrounding context is an assignment or the condition of an `if` statement. If it is a condition, then the expected type of the expression is Boolean. If it is an assignment, then the expected type is that of the identifier on the assignment's left-hand side. This identifier can be found two records below the current production in the attribute stack. ■

EXAMPLE 4.38

Semantic hooks for context

Semantic Hooks. To allow these cases to be treated uniformly, we can add *semantic hook*, or "marker" symbols to the grammar. Semantic hooks generate ε, and thus do not alter the language defined by the grammar; their only purpose is to hold inherited attributes:

$$\begin{array}{lll}
\mathit{stmt} & \longrightarrow & \texttt{id := } A\ \mathit{expr} \\
 & \longrightarrow & \ldots \\
 & \longrightarrow & \texttt{if } B\ \mathit{expr} \texttt{ then } \mathit{stmt} \\
A & \longrightarrow & \varepsilon\ \{\ \$\$.\mathsf{tp} := \${-}1.\mathsf{tp}\ \} \\
B & \longrightarrow & \varepsilon\ \{\ \$\$.\mathsf{tp} := \mathsf{Boolean}\ \} \\
\mathit{expr} & \longrightarrow & \ldots \{\ \mathsf{if}\ \$0.\mathsf{tp} = \mathsf{Boolean\ then}\ \ldots \}
\end{array}$$

■

Since the epsilon production for a semantic hook can provide an action routine, it is tempting to think of semantic hooks as a general technique to insert action routines in the middle of bottom-up productions. Unfortunately this is not the case: semantic hooks can be used only in places where the parser can be sure that it is in a given production. Placing a semantic hook anywhere else will break the "LR-ness" of the grammar, causing the parser generator to reject the modified grammar. Consider the following example:

EXAMPLE 4.39

Semantic hooks that break an LR CFG

$$\begin{array}{llll}
1. & \mathit{stmt} & \longrightarrow & \mathit{l\_val} \texttt{ := } \mathit{expr} \\
2. & & \longrightarrow & \texttt{id } \mathit{args} \\
3. & \mathit{l\_val} & \longrightarrow & \texttt{id } \mathit{quals}
\end{array}$$

4. *quals* ⟶ *quals* `.` `id`
5. ⟶ *quals* `(` *expr_list* `)`
6. ⟶ ε
7. *args* ⟶ `(` *expr_list* `)`
8. ⟶ ε

An *l-value* in this grammar is a "qualified" identifier: an identifier followed by optional array subscript and record field qualifiers.[2] We have assumed that the language follows the notation of Fortran and Ada, in which parentheses delimit both procedure call arguments and array subscripts. In the case of procedure calls, it would be natural to want an action routine to pass the symbol-table index of the subroutine into the argument list as an inherited attribute, so that it can be used to check the number and types of arguments:

stmt ⟶ `id` *A* *args*
A ⟶ ε { $$.proc_index := lookup ($0.name) }

If we try this, however, we will run into trouble, because the procedure call

```
foo(1, 2, 3);
```

and the array element assignment

```
foo(1, 2, 3) := 4;
```

begin with the same sequence of tokens. Until it sees the token after the closing parenthesis, the parser cannot tell whether it is working on production 1 or production 2. The presence of *A* in production 2 will therefore lead to a shift-reduce conflict; after seeing an `id`, the parser will not know whether to recognize *A* or shift `(`.

EXAMPLE 4.40
Action routines in the trailing part

Left Corners. In general, the right-hand side of a production in a context-free grammar is said to consist of the *left corner* and the *trailing part*. In the left corner we cannot be sure which production we are parsing; in the trailing part the production is uniquely determined. In an LL(1) grammar, the left corner is always empty. In an LR(1) grammar, it can consist of up to the entire right-hand side. Semantic hooks can safely be inserted in the trailing part of a production, but not in the left corner. `Yacc`/`bison` recognizes this fact explicitly by allowing action routines to be embedded in right-hand sides. It automatically converts the production

[2] In general, an l-value in a programming language is anything to which a value can be assigned (i.e., anything that can appear on the left-hand side of an assignment). From a low-level point of view, this is basically an address. An r-value is anything that can appear on the right-hand side of an assignment. From a low-level point of view, this is a value that can be stored at an address. We will discuss l-values and r-values further in Section 6.1.2.

$$S \longrightarrow \alpha\ \{\ \text{your code here}\ \}\ \beta$$

to

$$\begin{array}{lll} S & \longrightarrow & \alpha\ A\ \beta \\ A & \longrightarrow & \varepsilon\ \{\ \text{your code here}\ \} \end{array}$$

for some new, distinct symbol A. If the action routine is not in the trailing part, the resulting grammar will not be LALR(1), and `yacc/bison` will produce an error message. ■

EXAMPLE 4.41
Left factoring in lieu of semantic hooks

In our procedure call and array subscript example, we cannot place a semantic hook before the *args* of production 2 because this location is in the left corner. If we wish to look up a procedure name in the symbol table before we parse the arguments, we will need to combine the productions for statements that can begin with an identifier, in a manner reminiscent of the *left factoring* discussed in Section 2.3.2:

$$\begin{array}{lll} \textit{stmt} & \longrightarrow & \texttt{id}\ A\ \textit{quals}\ \textit{assign\_opt} \\ A & \longrightarrow & \varepsilon\ \{\ \text{\$\$.id\_index := lookup (\$0.name)}\ \} \\ \textit{quals} & \longrightarrow & \textit{quals}\ \texttt{.}\ \texttt{id} \\ & \longrightarrow & \textit{quals}\ \texttt{(}\ \textit{expr\_list}\ \texttt{)} \\ & \longrightarrow & \varepsilon \\ \textit{assign\_opt} & \longrightarrow & \texttt{:=}\ \textit{expr} \\ & \longrightarrow & \varepsilon \end{array}$$

This change eliminates the shift-reduce conflict, but at the expense of combining the entire grammar subtrees for procedure call arguments and array subscripts. To use the modified grammar we shall have to write action routines for *quals* that work for both kinds of constructs, and this can be a major nuisance. Users of LR-family parser generators often find that there is a tension between the desire for grammar clarity and parsability on the one hand and the need for semantic hooks to set inherited attributes on the other. ■

Top-Down Evaluation

Top-down parsers, as discussed in Chapter 2, come in two principal varieties: recursive descent and table driven. Attribute management in recursive descent parsers is almost trivial: inherited attributes of symbol *foo* take the form of parameters passed into the parsing routine named foo; synthesized attributes are the return parameters. These synthesized attributes can then be passed as inherited attributes to symbols later in the current production, or returned as synthesized attributes of the current left-hand side.

Attribute space management for automatically generated top-down parsers is somewhat more complex. Because they allow action routines at arbitrary locations in a right-hand side, top-down parsers avoid the need to modify the grammar in order to insert semantic hooks. (Of course, because they must have empty left corners, top-down grammars can be harder to write in the first place.) Because the parse stack describes the future, instead of the past, we cannot employ an attribute

$E \longrightarrow T \; \{ \text{ TT.st := T.val } \}^{\mathbf{1}} \; TT \; \{ \text{ E.val := TT.val } \}^{\mathbf{2}}$

$TT_1 \longrightarrow \texttt{+} \; T \; \{ \text{ TT}_2\text{.st := TT}_1\text{.st} + \text{T.val } \}^{\mathbf{3}} \; TT_2 \; \{ \text{ TT}_1\text{.val := TT}_2\text{.val } \}^{\mathbf{4}}$

$TT_1 \longrightarrow \texttt{-} \; T \; \{ \text{ TT}_2\text{.st := TT}_1\text{.st} - \text{T.val } \}^{\mathbf{5}} \; TT_2 \; \{ \text{ TT}_1\text{.val := TT}_2\text{.val } \}^{\mathbf{6}}$

$TT \longrightarrow \varepsilon \{ \text{ TT.val := TT.st } \}^{\mathbf{7}}$

$T \longrightarrow F \; \{ \text{ FT.st := F.val } \}^{\mathbf{8}} \; FT \; \{ \text{ T.val := FT.val } \}^{\mathbf{9}}$

$FT_1 \longrightarrow \texttt{*} \; F \; \{ \text{ FT}_2\text{.st := FT}_1\text{.st} \times \text{F.val } \}^{\mathbf{10}} \; FT_2 \; \{ \text{ FT}_1\text{.val := FT}_2\text{.val } \}^{\mathbf{11}}$

$FT_1 \longrightarrow \texttt{/} \; F \; \{ \text{ FT}_2\text{.st := FT}_1\text{.st} \div \text{F.val } \}^{\mathbf{12}} \; FT_2 \; \{ \text{ FT}_1\text{.val := FT}_2\text{.val } \}^{\mathbf{13}}$

$FT \longrightarrow \varepsilon \{ \text{ FT.val := FT.st } \}^{\mathbf{14}}$

$F_1 \longrightarrow \texttt{-} \; F_2 \; \{ \text{ F}_1\text{.val :=} - \text{F}_2\text{.val } \}^{\mathbf{15}}$

$F \longrightarrow \texttt{(} \; E \; \texttt{)} \; \{ \text{ F.val := E.val } \}^{\mathbf{16}}$

$F \longrightarrow \texttt{const} \; \{ \text{ F.val := C.val } \}^{\mathbf{17}}$

Figure 4.26 LL(1) grammar for constant expressions, with action routines. The boldface superscripts are for reference in Figure C-4.27.

stack that simply mirrors the parse stack. Our two principal options are to equip the parser with a (more complicated) algorithm for automatic space management, or to require action routines to manage space explicitly.

Automatic Management. Automatic management of attribute space for top-down parsing is more complicated than it is for bottom-up parsing. It is also more space intensive. We can still use an attribute stack, but it has to contain all of the symbols in all of the productions between the root of the (hypothetical) parse tree and the current point in the parse. All of the right-hand side symbols of a given production are adjacent in the stack; the left-hand side is buried in the right-hand side of a deeper (closer to the root) production.

EXAMPLE 4.42
Operation of an LL attribute stack

Figure C-4.26 contains an LL(1) grammar for constant expressions, with action routines. Figure C-4.27 uses this grammar to trace the operation of a top-down attribute stack on the sample input `(1 + 3) * 2`. The left-hand column shows the parse stack. The right-hand column shows the attribute stack. Three global pointers index into the attribute stack. One (shown as an "arrow-boxed" `L` in the trace) identifies the record in the attribute stack that holds the attributes of the left-hand side symbol of the current production. The second (shown as an arrow-boxed `R` in the trace) identifies the first symbol on the right-hand side of the production. `L` and `R` allow the action routines to find the attributes of the symbols of the current production. The third pointer (shown as an arrow-boxed `N` in the trace) identifies the first symbol within the right-hand side that has not yet been completely parsed. It allows the parser to update `L` correctly when a production is predicted.

At any given time, the attribute stack contains all symbols of all productions on the path between the root of the parse tree and the symbol currently at the top of the parse stack. Figure C-4.28 identifies these symbols graphically at the point in Figure C-4.27 immediately above the eight elided lines. Symbols to the left in the parse tree have already been reclaimed; those to the right have yet to be allocated.

`E$`	$\boxed{N}\ E_?$
`T1TT2:$`	$\boxed{L}\ E_?\ \boxed{R}\boxed{N}\ T_?\ TT_{?,?}$
`F8FT9:1TT2:$`	$E_?\ \boxed{L}\ T_?\ TT_{?,?}\ \boxed{R}\boxed{N}\ F_?\ FT_{?,?}$
`(E) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ \boxed{L}\ F_?\ FT_{?,?}\ \boxed{R}\boxed{N}\ (\ E_?\ )$
`E) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ \boxed{L}\ F_?\ FT_{?,?}\ \boxed{R}\ (\ \boxed{N}\ E_?\ )$
`T1TT2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ \boxed{L}\ E_?\ )\ \boxed{R}\boxed{N}\ T_?\ TT_{?,?}$
`F8FT9:1TT2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ \boxed{L}\ T_?\ TT_{?,?}\ \boxed{R}\boxed{N}\ F_?\ FT_{?,?}$
`C17:8FT9:1TT2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_?\ TT_{?,?}\ \boxed{L}\ F_?\ FT_{?,?}\ \boxed{R}\boxed{N}\ C_1$
`17:8FT9:1TT2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_?\ TT_{?,?}\ \boxed{L}\ F_?\ FT_{?,?}\ \boxed{R}\ C_1\ \boxed{N}$
`:8FT9:1TT2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_?\ TT_{?,?}\ \boxed{L}\ F_1\ FT_{?,?}\ \boxed{R}\ C_1\ \boxed{N}$
`8FT9:1TT2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ \boxed{L}\ T_?\ TT_{?,?}\ \boxed{R}\ F_1\ \boxed{N}\ FT_{?,?}$
`FT9:1TT2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ \boxed{L}\ T_?\ TT_{?,?}\ \boxed{R}\ F_1\ \boxed{N}\ FT_{1,?}$
`14:9:1TT2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_?\ TT_{?,?}\ F_1\ \boxed{L}\ FT_{1,?}\ \boxed{R}\boxed{N}$
`:9:1TT2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_?\ TT_{?,?}\ F_1\ \boxed{L}\ FT_{1,1}\ \boxed{R}\boxed{N}$
`9:1TT2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ \boxed{L}\ T_?\ TT_{?,?}\ \boxed{R}\ F_1\ FT_{1,1}\ \boxed{N}$
`:1TT2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ \boxed{L}\ T_1\ TT_{?,?}\ \boxed{R}\ F_1\ FT_{1,1}\ \boxed{N}$
`1TT2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ \boxed{L}\ E_?\ )\ \boxed{R}\ T_1\ \boxed{N}\ TT_{?,?}$
`TT2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ \boxed{L}\ E_?\ )\ \boxed{R}\ T_1\ \boxed{N}\ TT_{1,?}$
`+T3TT4:2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ \boxed{L}\ TT_{1,?}\ \boxed{R}\boxed{N}\ +\ T_?\ TT_{?,?}$
`T3TT4:2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ \boxed{L}\ TT_{1,?}\ \boxed{R}\ +\ \boxed{N}\ T_?\ TT_{?,?}$
`F8FT9:3TT4:2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ TT_{1,?}\ +\ \boxed{L}\ T_?\ TT_{?,?}\ \boxed{R}\boxed{N}\ F_?\ FT_{?,?}$
`C17:8FT9:3TT4:2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ TT_{1,?}\ +\ T_?\ TT_{?,?}\ \boxed{L}\ F_?\ FT_{?,?}\ \boxed{R}\boxed{N}\ C_3$
⟨ eight lines omitted ⟩	
`3TT4:2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ \boxed{L}\ TT_{1,?}\ \boxed{R}\ +\ T_3\ \boxed{N}\ TT_{?,?}$
`TT4:2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ \boxed{L}\ TT_{1,?}\ \boxed{R}\ +\ T_3\ \boxed{N}\ TT_{4,?}$
`7:4:2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ TT_{1,?}\ +\ T_3\ \boxed{L}\ TT_{4,?}\ \boxed{R}\boxed{N}$
`:4:2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ TT_{1,?}\ +\ T_3\ \boxed{L}\ TT_{4,4}\ \boxed{R}\boxed{N}$
`4:2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ \boxed{L}\ TT_{1,?}\ \boxed{R}\ +\ T_3\ TT_{4,4}\ \boxed{N}$
`:2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ E_?\ )\ T_1\ \boxed{L}\ TT_{1,4}\ \boxed{R}\ +\ T_3\ TT_{4,4}\ \boxed{N}$
`2:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ \boxed{L}\ E_?\ )\ \boxed{R}\ T_1\ TT_{1,4}\ \boxed{N}$
`:) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_?\ FT_{?,?}\ (\ \boxed{L}\ E_4\ )\ \boxed{R}\ T_1\ TT_{1,4}\ \boxed{N}$
`) 16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ \boxed{L}\ F_?\ FT_{?,?}\ \boxed{R}\ (\ E_4\ \boxed{N}\ )$
`16:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ \boxed{L}\ F_?\ FT_{?,?}\ \boxed{R}\ (\ E_4\ )\ \boxed{N}$
`:8FT9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ \boxed{L}\ F_4\ FT_{?,?}\ \boxed{R}\ (\ E_4\ )\ \boxed{N}$
`8FT9:1TT2:$`	$E_?\ \boxed{L}\ T_?\ TT_{?,?}\ \boxed{R}\ F_4\ \boxed{N}\ FT_{?,?}$
`FT9:1TT2:$`	$E_?\ \boxed{L}\ T_?\ TT_{?,?}\ \boxed{R}\ F_4\ \boxed{N}\ FT_{4,?}$
`*F10FT11:9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_4\ \boxed{L}\ FT_{4,?}\ \boxed{R}\boxed{N}\ *\ F_?\ FT_{?,?}$
`F10FT11:9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_4\ \boxed{L}\ FT_{4,?}\ \boxed{R}\ *\ \boxed{N}\ F_?\ FT_{?,?}$
`C17:10FT11:9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_4\ FT_{4,?}\ *\ \boxed{L}\ F_?\ FT_{?,?}\ \boxed{R}\boxed{N}\ C_2$
`17:10FT11:9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_4\ FT_{4,?}\ *\ \boxed{L}\ F_?\ FT_{?,?}\ \boxed{R}\ C_2\ \boxed{N}$
`:10FT11:9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_4\ FT_{4,?}\ *\ \boxed{L}\ F_2\ FT_{?,?}\ \boxed{R}\ C_2\ \boxed{N}$
`10FT11:9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_4\ \boxed{L}\ FT_{4,?}\ \boxed{R}\ *\ F_2\ \boxed{N}\ FT_{?,?}$
`FT11:9:1TT2:$`	$E_?\ T_?\ TT_{?,?}\ F_4\ \boxed{L}\ FT_{4,?}\ \boxed{R}\ *\ F_2\ \boxed{N}\ FT_{8,?}$
⟨ six lines omitted ⟩	
`1TT2:$`	$\boxed{L}\ E_?\ \boxed{R}\ T_8\ \boxed{N}\ TT_{?,?}$
`TT2:$`	$\boxed{L}\ E_?\ \boxed{R}\ T_8\ \boxed{N}\ TT_{8,?}$
`7:2:$`	$E_?\ T_8\ \boxed{L}\ TT_{8,?}\ \boxed{R}\boxed{N}$
`:2:$`	$E_?\ T_8\ \boxed{L}\ TT_{8,8}\ \boxed{R}\boxed{N}$
`2:$`	$\boxed{L}\ E_?\ \boxed{R}\ T_8\ TT_{8,8}\ \boxed{N}$
`:$`	$\boxed{L}\ E_8\ \boxed{R}\ T_8\ TT_{8,8}\ \boxed{N}$
`$`	$E_8\ \boxed{N}$

Figure 4.27 Trace of the parse stack (left) and attribute stack (right) for `(1 + 3) * 2`, using the grammar (and action routine numbers) of Figure C-4.26. Subscripts in the attribute stack indicate the values of attributes. For symbols with two attributes, st comes first.

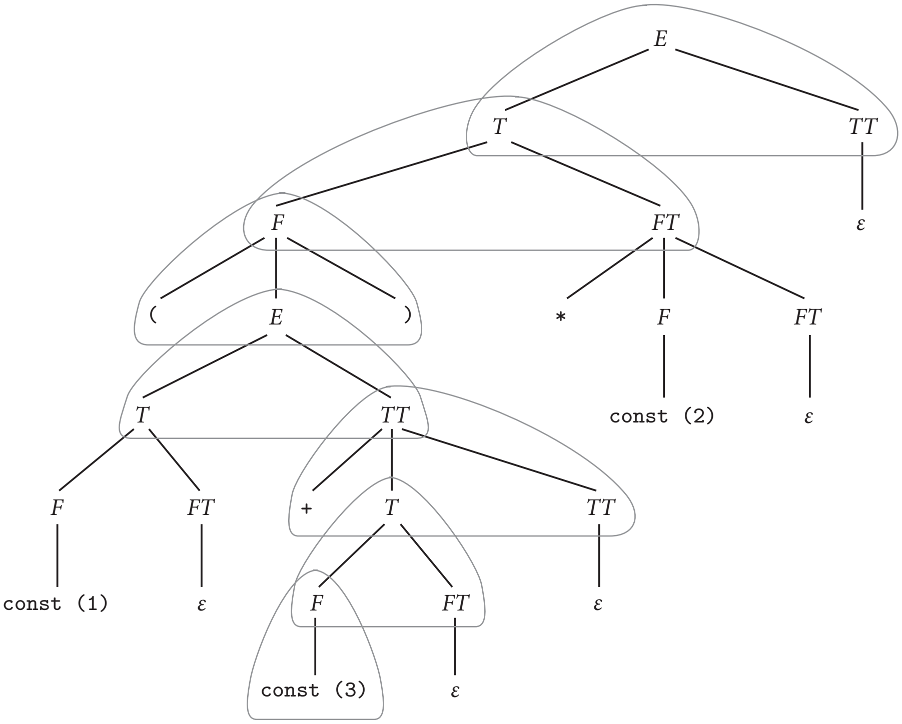


Figure 4.28 Productions with symbols currently in the attribute stack during a parse of (1 + 3) * 2 (using the grammar of Figure C-4.26), at the point where we are about to parse the 3. In Figure C-4.27 this point corresponds to the line immediately above the eight elided lines.

At start-up, the attribute stack contains a record for the start symbol, pointed at by N. When we push the right-hand side of a predicted production onto the parse stack, we add an "end-of-production" marker, represented by a colon in the trace. At the same time, we push records for the right-hand-side symbols onto the attribute stack. (These are *added* to the attribute stack; they do not replace the left-hand side.) Prior to pushing these entries, we save the current L and R pointers in another stack (not shown). We then set L to the old N, and make R and N point to the newly pushed right-hand side.

When we see an action symbol at the top of the parse stack (shown in the trace as a small bold number), we pop it and execute the corresponding action routine. When we match a terminal at the top of the parse stack, we pop it and move N forward one record in the attribute stack. When we see an end-of-production marker at the top of the parse stack, we pop it, set N to the attribute record following the one currently pointed at by L, pop everything from R forward off of the attribute stack, and restore the most recently saved values of L and R. ■

$E \longrightarrow T\ TT$
$TT \longrightarrow$ + T { bin_op ("+") } TT
$TT \longrightarrow$ - T { bin_op ("−") } TT
$TT \longrightarrow \varepsilon$
$T \longrightarrow F\ FT$
$FT \longrightarrow$ * F { bin_op ("×") } FT
$FT \longrightarrow$ / F { bin_op ("÷") } FT
$FT \longrightarrow \varepsilon$
$F \longrightarrow$ - F { un_op ("$^{+}/_{-}$") }
$F \longrightarrow$ (E)
$F \longrightarrow$ `const` { push_leaf (cur_tok.val) }

Figure 4.29 Ad hoc management of attribute space in an LL(1) grammar to build a syntax tree.

It should be emphasized that while the trace is long and tedious, its complexity is completely hidden from the writer of action routines. Once the space management routines are integrated with the driver for a top-down parser generator, all the compiler writer sees is the grammar of Figure C-4.26. When the compiler writer refers to attributes of the symbol on the left-hand side of a production, the parser generator will access entry L in the attribute stack; when the compiler writer refers to attributes of the kth symbol on the right-hand side, the parser generator will access entry $\mathtt{R} - k - 1$. In comparing Figures C-4.25 and C-4.27, one should also note that reduction and execution of a production's action routine are shown as a single step in the LR trace; they are shown separately in the LL trace, making that trace appear more complex than it really is.

Ad Hoc Management. One drawback of automatic space management for top-down grammars is the frequency with which the compiler writer must specify copy routines. Of the 17 action routines in Figure C-4.26, 12 simply move information from one place to another. The time required to execute these routines can be minimized by copying pointers, rather than large records, but compiler writers may still consider the copies a nuisance.

EXAMPLE 4.43
Ad hoc management of a semantic stack

An alternative is to manage space explicitly within the action routines, pushing and popping an ad hoc *semantic stack* only when information is generated or consumed. Using this technique, we can replace the action routines of Figure C-4.26 with the simpler version shown in Figure C-4.29. Variable cur_tok is assumed to contain the synthesized attributes of the most recently matched token. The semantic stack contains pointers to syntax tree nodes. The push_leaf routine creates a node for a specified constant and pushes a pointer to it onto the semantic stack. The un_op routine pops the top pointer off the stack, makes it the child of a newly created node for the specified unary operator, and pushes a pointer to that node back on the stack. The bin_op routine pops the top *two* pointers off the semantic stack and pushes a pointer to a newly created node for the specified binary operator.

When the parse of *E* is completed, a pointer to a syntax tree describing its yield will be found in the top-most record on the semantic stack.

The advantage of ad hoc space management is clearly the smaller number of rules and the elimination of the inherited attributes used to represent left context. The disadvantage is that the compiler writer must be aware of what is in the semantic stack at all times, and must remember to push and pop it when appropriate.

One further advantage of an ad hoc semantic stack is that it allows action routines to push or pop an arbitrary number of records. With automatic space management, the number of records that can be seen by any one routine is limited by the number of symbols in the current production. The difference is particularly important in the case of productions that generate lists. In Example C-4.36 we saw an SLR(1) grammar for declarations in the style of C and Fortran, in which the type name precedes the list of identifiers. Here is an LL(1) grammar fragment for a language in the style of Pascal and Ada, in which the variables precede the type:

EXAMPLE 4.44
Processing lists with an attribute stack

```
dec ⟶ id_list : type
id_list ⟶ id id_list_tail
id_list_tail ⟶ , id_list
             ⟶ ε
```

Without resorting to non-L-attributed flow (see Exercise C-4.41), we cannot pass the declared type into *id_list* as an inherited attribute. Instead, we must save up the list of identifiers and enter them into the symbol table *en masse* when the type is finally encountered. With automatic management of space for attributes, the action routines would look something like this:

```
dec ⟶ id_list : type { declare_vars(id_list.chain, type.tp) }
id_list ⟶ id id_list_tail { id_list.chain := append(id.name, id_list_tail.chain) }
id_list_tail ⟶ , id_list { id_list_tail.chain := id_list.chain }
             ⟶ ε { id_list_tail.chain := null }
```

EXAMPLE 4.45
Processing lists with a semantic stack

With ad hoc management of space, we can get by without the linked list:

```
dec ⟶ { push(marker) }
      id_list : type
      {  pop(tp)
         pop(name)
         while name ≠ marker
               declare_var(name, tp)
               pop(name)  }
id_list ⟶ id { push(cur_tok.name) } id_list_tail
id_list_tail ⟶ , id_list
             ⟶ ε
```

Neither automatic nor ad hoc management of attribute space in top-down parsers is clearly superior to the other. The ad hoc approach eliminates the need for many copy rules and inherited attributes, and is consequently somewhat more time and space efficient. It also allows lists to be embedded in the semantic stack. On the other hand, it requires that the programmer who writes the action routines be continually aware of what is in the stack and why, in order to push and pop it appropriately. In the final analysis, the choice is an engineering tradeoff driven by the particular needs of the project.

✓ CHECK YOUR UNDERSTANDING

49. Explain how to manage space for synthesized attributes in a bottom-up parser.
50. Explain how to manage space for inherited attributes in a bottom-up parser.
51. Define *left corner* and *trailing part*.
52. Under what circumstances can an action routine be embedded in the right-hand side of a production in a bottom-up parser? Equivalently, under what circumstances can a marker symbol be embedded in a right-hand side without rendering the grammar non-LR?
53. Summarize the tradeoffs between automatic and ad hoc management of space for attributes in a top-down parser.
54. At any given point in a top-down parse, which symbols will have attribute records in an automatically managed attribute stack?

4 Program Semantics

4.8 Exercises

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

4.23 Write an S-attributed attribute grammar, based on the CFG of Example C-4.28, that accumulates the value of the overall expression into the root of the tree. You will need to use dynamic memory allocation so that individual attributes can hold an arbitrary amount of information.

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

Using the underlying CFG of Figure C-4.16, write an attribute grammar that will associate with the root of the parse tree a sequence of postfix calculator button pushes, seq, that will compute the arithmetic value of the tokens derived from that symbol. You may assume the existence of a function buttons(c) that returns a sequence of button pushes (ending with $\boxed{\text{E}}$ on a postfix calculator) for the constant `c`. You may also assume the existence of a concatenation function for sequences of button pushes.

4.25 Repeat the previous exercise using the underlying CFG of Figure C-4.18.

4.26 Consider the following grammar for reverse Polish arithmetic expressions:

$$
\begin{aligned}
&E \longrightarrow E\ E\ op \mid \texttt{id}\\
&op \longrightarrow \texttt{+} \mid \texttt{-} \mid \texttt{*} \mid \texttt{/}
\end{aligned}
$$

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

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

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

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

$$
\begin{aligned}
&C \longrightarrow digits\ \texttt{.}\ digits\\
&digits \longrightarrow digit\ more\_digits\\
&more\_digits \longrightarrow digits \mid \varepsilon\\
&digit \longrightarrow \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9}
\end{aligned}
$$

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

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

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

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

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

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

4.32 Write an LL(1) grammar with action routines and automatic attribute space management that generates the reverse Polish translation described in Exercise C-4.24.

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

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

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

4.35 Rewrite the attribute grammar of Figure C-4.23 in the form of an ad hoc tree traversal consisting of mutually recursive subroutines in your favorite programming language. Keep the symbol table in a global variable, rather than passing it through arguments.

4.36 Augment the attribute grammar of Figure C-4.20, Figure C-4.21 to initialize a synthesized attribute in every syntax tree node that indicates the location (line and column) at which the corresponding construct appears in the source program. You may assume that the scanner initializes the location of every token.

4.37 Modify the CFG and attribute grammar of Figures 4.1 and C-4.23 to permit mixed integer and real expressions, without the need for `float` and `trunc`. You will want to add an annotation to any node that must be coerced to the opposite type, so that the code generator will know to generate code to do so. Be sure to think carefully about your coercion rules. In the expression `my_int + my_real`, for example, how will you know whether to coerce the integer to be a real, or to coerce the real to be an integer?

4.38 A potential objection to the abstract attribute grammar of Example C-4.33 is that it repeatedly copies the entire symbol table from one node to another. In this particular tiny language, it is easy to see that the referencing environment never shrinks: the symbol table changes only with the addition of new identifiers. Exploiting this observation, show how to modify the pseudocode of Figure C-4.23 so that it copies only pointers, rather than the entire symbol table.

4.39 Your solution to the previous exercise probably doesn't generalize to languages with nontrivial scoping rules. Explain how an AG such as that in Figure C-4.23 might be modified to use a global symbol table similar to the one described in Section C-3.4.1. Among other things, you should consider nested scopes, the hiding of names in outer scopes, and the requirement (not enforced by the table of Section C-3.4.1) that variables be declared before they are used.

4.40 Repeat Exercise C-4.32 using ad hoc attribute space management. Instead of accumulating the translation into a data structure, write it to a file on the fly.

4.41 Rewrite the grammar for declarations of Example C-4.44 without the requirement that your attribute flow be L-attributed. Try to make the grammar as simple and elegant as possible (you shouldn't need to accumulate lists of identifiers).

4.42 Fill in the missing lines in Figure C-4.27.

4.43 Consider the following grammar with action routines:

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

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

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

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

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

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

If the attribute stack is of finite size, it is guaranteed to overflow for some long but valid block of straight-line code. The problem is especially unfortunate since, with the exception of the accumulated output code, none of the repeated symbols in the attribute stack contains any useful attributes once its substructure has been parsed.

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

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

4 Program Semantics

4.9 Explorations

4.50 One of the most influential applications of attribute grammars was the Cornell Synthesizer Generator [Rep84, RT88]. Learn how the Generator used attribute grammars not only for incremental update of semantic information in a program under edit, but also for automatic creation of language based editors from formal language specifications. How general is this technique? What applications might it have beyond syntax-directed editing of computer programs?

4.51 The attribute grammars used in this chapter are all quite simple. Most are S- or L-attributed. All are noncircular. Are there any practical uses for more complex attribute grammars? How about automatic attribute evaluators? Using the Bibliographic Notes as a starting point, conduct a survey of attribute evaluation techniques. Where is the line between practical techniques and intellectual curiosities?

4.52 As described in Section C-4.6.4, `yacc/bison` will refuse to accept action routines in the left corner of a production. Is there any way around this problem? Can you imagine implementing an extended version of the tool that would permit action routines in arbitrary locations? What would be the challenges? The cost?

4.53 Learn how attribute space is managed in the ANTLR parser generator. How does it compare to the techniques described in Section C-4.6.4 for top-down parsing?

5 Target Machine Architecture

Processor implementations change over time, as people invent better ways of doing things, and as technological advances (e.g., increases in the number of transistors that will fit on one chip) make things feasible that were not feasible before. Processor architectures also change, for at least two reasons. Some technological advances can be exploited only by changing the hardware/software interface—for example by increasing the number of bits that can be added or multiplied in a single instruction. In addition, experience with compilers and applications sometimes suggests that certain new instructions would make programs simpler or faster.

Occasionally, technological and intellectual trends converge to produce a revolutionary change in both architecture and implementation. We will discuss four such changes in Section C-5.4: the development of microprogramming in the early 1960s, the development of the microprocessor in the early to mid-1970s, the development of reduced instruction set computing (RISC) in the early 1980s, and the move to multithreaded and multicore processors in the first decade of the 21st century. A fifth major change has occurred with the proliferation of general-purpose graphical processing units (GPUs) and other accelerators; these are beyond the scope of this text.

This chapter provides a quick overview of those aspects of computer architecture most essential to the task of compiler construction. In Sections C-5.1–C-5.3 we consider the hierarchical organization of memory, the types (formats) of data found in memory, and the instructions used to manipulate those data. The coverage is necessarily somewhat cursory and high-level; much more detail can be found in books on computer architecture or organization (e.g., Chapters 2–5 of Patterson and Hennessy's outstanding text [PH20]).[1]

[1] John L. Hennessey (1952–) and David A. Patterson (1947–) are primarily known for work in computer architecture, but have also made important contributions to programming languages and implementation techniques. Hennessey is currently Chairman of Alphabet, Inc. (the parent company of Google) and was previously President of Stanford University. Patterson is a Professor at the University of California, Berkeley; he also served as President of the Association for Computing Machinery (ACM) from 2004–2006. Among many other accomplishments, Hennessey and Patterson pioneered the design of reduced instruction set computers (RISC), for which they shared the ACM Turing Award in 2017.

	Typical access time	**Typical capacity**
Registers	0.2–0.4 ns	0.5–3 K bytes
Primary (L1) cache	0.4–1 ns	32 K–256 K bytes
last-level (typically L3) cache	4–40 ns	1–32 M bytes
Main memory	80–250 ns	1 G byte to 1 T byte
Flash (SSD)	10–40 μs	120 G bytes to 16 T bytes
Disk (HDD)	3–10 ms	1–16 T bytes
Tape	1–50 s	up to 30 T bytes per cartridge

Figure 5.1 **The memory hierarchy of a workstation-class computer.** Access times and capacities are approximate, based on 2023 technology. Registers are accessed within a single clock cycle; primary cache takes 2 to 4 cycles. Main memory typically resides on the far side of a bus or other communication channel and is consequently slower. Flash times vary with manufacturing technology, and are longer for writes than reads. Disk and tape times are constrained by the movement of physical parts.

We consider the interplay between architecture and implementation in Section C-5.4. As illustrative examples, we consider the widely used x86 and Arm instruction sets. Finally, in Section C-5.5, we consider some of the issues that make compiling for modern processors a challenging task.

5.1 The Memory Hierarchy

Memory on most machines consists of a numbered sequence of 8-bit bytes. The size of the sequence—the number of distinct locations—is limited by the number of bits used to represent an address. This is a sufficiently important number that it is often used to categorize machines. Programs on a "32-bit machine" can address no more than 2^{32} bytes (4 GB) of memory. Programs on a "64-bit machine" can (at least in principle) address 4 billion times as much.

EXAMPLE 5.1

Memory hierarchy stats

It is not uncommon for modern workstations to contain tens of gigabytes of memory—much too much to fit on the same chip as the processor. The time it takes to reach memory depends on its distance from the processor. Off-chip memory—particularly when located on the other side of an interconnection network shared by other processors and devices—is particularly slow. Most computers therefore employ a *memory hierarchy*, in which things that are used more often are kept close at hand. A typical memory hierarchy, with access times and capacities, is shown in Figure C-5.1. ■

Only three of the levels of the memory hierarchy—registers, memory, and devices—are a visible part of the hardware/software interface. Compilers manage registers explicitly, loading them from memory when needed and storing them back to memory when done, or when the registers are needed for something else. Caches are managed by the hardware. Devices are generally accessed only by the operating system.

Registers hold small amounts of data that can be accessed very quickly. A typical modern machine has two sets of registers—one to hold integer operands, the other floating-point. Additional sets may be used for special purposes—for example, vector instructions, which operate, in parallel, on a sequence of shorter values packed into a longer register. There are usually several special-purpose registers as well, including the *program counter* (PC) and the *processor status register*. The program counter holds the address of the next instruction to be executed. It is incremented automatically when fetching most instructions; branches work by changing it explicitly. The processor status register contains a variety of bits of importance to the operating system (privilege level, interrupt priority level, trap enable bits) and, on some machines, a few bits of importance to the compiler writer. Principal among these are *condition codes*, which indicate whether the most recent arithmetic or logical operation resulted in a zero, a negative value, and/or arithmetic overflow. (We will consider condition codes in more detail in Section C-5.3.2.)

Because registers can be accessed every cycle, while memory, generally, cannot, good compilers expend a great deal of effort trying to make sure that the data they need most often are in registers, and trying to minimize the amount of time spent moving data back and forth between registers and memory. We will consider algorithms for register management in Section C-5.5.2.

Caches are generally smaller but faster than main memory. They are designed to exploit *locality*: the tendency of most computer programs to access the same or nearby locations in memory repeatedly. By automatically moving the contents of these locations into cache, a hierarchical memory system can dramatically improve performance. The idea makes intuitive sense: loops tend to access the same local variables in every iteration, and to walk sequentially through arrays. Instructions, likewise, tend to be loaded from consecutive locations, and code that accesses one element of a structure (or member of a class) is likely to access another.

Cache architecture varies quite a bit across machines. Primary caches, also known as *level-1 (L1) caches*, are invariably located on the same chip as the processor, and usually come in pairs: one for instructions (the L1 I-cache) and another for data (the L1 D-cache), both of which can be accessed every cycle. Secondary (L2) and tertiary (L3) caches are larger and slower, but still faster than main memory.

DESIGN & IMPLEMENTATION

5.2 The processor/memory gap

For roughly 50 years, from the 1950s until about 2004, processor speed increased much faster than memory speed. As a result, the number of processor cycles required to access memory grew dramatically, and caches became increasingly critical to performance. To improve the effectiveness of caching, programmers need to choose algorithms whose data access patterns have a high degree of locality. High-quality compilers, likewise, need to consider locality of access when choosing among the many possible translations of a given program.

In a modern desktop or laptop system they are typically also on the same chip as the processor.

Most processors today have more than one processing *core* on a single chip. L1 caches are almost always private to a single core. L2 caches may be private or shared by 2–4 cores. L3 caches are generally shared by all cores on a chip. Caches are managed entirely in hardware on most machines, but compilers can increase their effectiveness by generating code with a high degree of locality.

A memory access that finds its data in the cache is said to be a *cache hit*. An access that does not find its data in the cache is said to be a *cache miss*. On a miss, the hardware automatically loads a *line* of the cache with a contiguous block of data containing the requested location, obtained from the next lower level of cache or main memory. Cache lines vary from as few as 16 to as many as 512 bytes in length. Assuming that the cache was already full, the load will displace some other line, which is written back to memory if it has been modified.

A final characteristic of memory that is important to the compiler is known as data *alignment*. Most machines are able to manipulate operands of several sizes—typically one, two, four, or eight bytes. Most modern instruction sets refer to these as byte, half-word, word, and double-word operands, respectively; on the x86 they are byte, word, double-word, and quad-word operands. Many recent architectures require n-byte operands to appear in memory at addresses that are evenly divisible by n (at least for $n \leq 4$). A 4-byte integer, for example, must typically appear at a location whose address is evenly divisible by four. This restriction occurs for two reasons. First, buses are designed in such a way that data are delivered to the processor over bit-parallel, aligned communication paths. Loading an integer from an odd address would require that the bits be shifted, adding logic (and time) to the load path. The x86 and Arm, which allow most operands to appear at arbitrary addresses, run faster if those operands are properly aligned. Second, on machines with fixed-size instructions, there are generally not enough bits to specify both an operation (e.g., load) and a full address. As we shall see in Section C-5.3.1, it is typical to specify an address in terms of an *offset* from some *base location* specified by a register. Requiring that integers be word-aligned allows the offset to be specified in words, rather than in bytes, quadrupling the amount of memory that can be accessed using offsets from a given base register.

5.2 Data Representation

Data in the memory of most computers are untyped: bits are simply bits. *Operations* are typed, in the sense that different operations *interpret* the bits in memory in different ways. Typical *data formats* include instructions, addresses, binary integers of various lengths, floating-point (real) numbers of various lengths, and characters.

EXAMPLE 5.2

Big- and little-endian

Integers typically come in half-word, word, and double-word lengths. Floating-point numbers typically come in word and double-word lengths, commonly referred to as *single*- and *double-precision*. Some machines store the least-significant

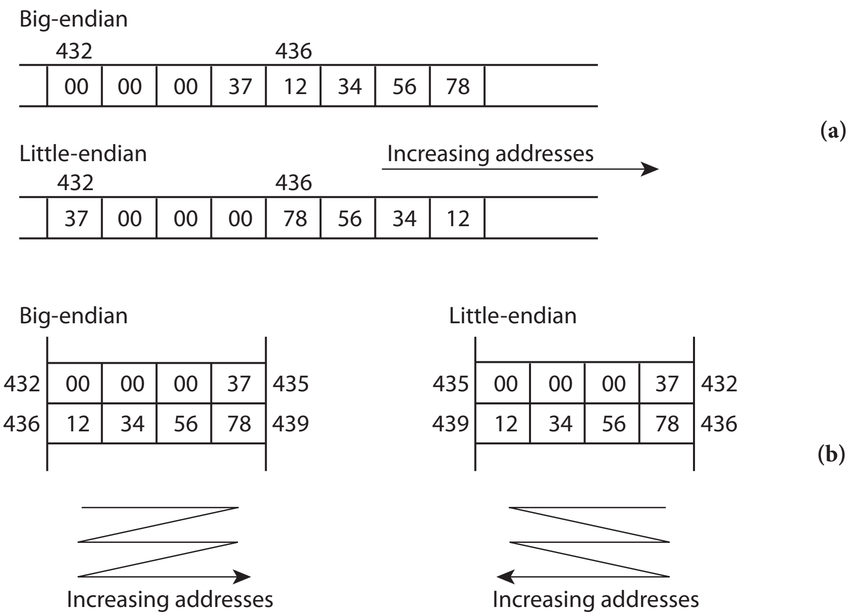


Figure 5.2 **Big-endian and little-endian byte orderings.** (a) Two 4-byte quantities, the numbers 37_{16} and $12\,34\,56\,78_{16}$, stored at addresses 432 and 436, respectively. (b) The same situation, with memory visualized as a byte-addressable array of words.

byte of a multiword datum at the address of the datum itself, with bytes of increasing numeric significance at higher-numbered addresses. Other machines store the bytes in the opposite order. The first option is called *little-endian*; the second is called *big-endian*. In either case, an n-byte datum stored at address t occupies bytes t through $t + n - 1$. The advantage of a little-endian organization is that it is tolerant of variations in operand size. If the value 37 is stored as a word and then a byte is read from the same location, the value 37 will be returned. On a big-endian machine, the value 0 will be returned (the upper eight bits of the number 37, when stored in 32 bits). The problem with the little-endian approach is that it seems to scramble the bytes of integers, when read from left to right (see Figure C-5.2a). Little-endian-ness makes a bit more sense if one thinks of memory as a (byte-addressable) array of words (Figure C-5.2b). The x86 is little-endian. IBM's z Series (mainframe) machines are big-endian. Most other common processors, including the Arm, MIPS, Power, and RISC-V families, can run in either mode, at the choice of the operating system.

Support for characters varies widely. A few machines can perform arbitrary arithmetic and logical operations on 1-byte quantities. Most can load and store bytes from or to memory, but operate only on longer quantities in registers. Some legacy machines, including the x86, provide instructions that perform operations on strings of characters, such as copying, comparing, or searching. On more

0 0 0 0	0	1 0 0 0	8
0 0 0 1	1	1 0 0 1	9
0 0 1 0	2	1 0 1 0	a
0 0 1 1	3	1 0 1 1	b
0 1 0 0	4	1 1 0 0	c
0 1 0 1	5	1 1 0 1	d
0 1 1 0	6	1 1 1 0	e
0 1 1 1	7	1 1 1 1	f

Figure 5.3 The hexadecimal digits.

modern machines (again including the x86), vector instructions can also be used to operate on strings.

5.2.1 Integer Arithmetic

Binary integers are almost universally represented in two related formats: straightforward binary place-value for unsigned numbers, and *two's complement* for signed numbers. An n-bit unsigned integer has a value in the range $0 \ldots 2^n - 1$, inclusive. An n-bit two's complement integer has a value in the range $-2^{n-1} \ldots 2^{n-1} - 1$, inclusive. Most instruction sets provide two forms of most of the arithmetic operators: one for unsigned numbers and one for signed numbers. Even for languages in which integers are always signed, unsigned arithmetic is important for the manipulation of addresses (e.g., pointers).

EXAMPLE 5.3
Hexadecimal numbers

An n-bit unsigned integer with binary representation $b_{n-1}\, b_{n-2}\, \ldots\, b_2\, b_1\, b_0$ has the value $\sum_{0 \le i < n} b_i 2^i$. Because the bit pattern corresponding to a given decimal value is non-obvious, and because bit patterns written as strings of 0's and 1's are cumbersome, computer scientists commonly represent integer values in *hexadecimal*, or base-16 notation. Hexadecimal uses the letters a to f as six additional digits, representing the values 10 to 15 in decimal (see Figure C-5.3). Because $2^4 = 16$, every digit in a hexadecimal number corresponds to exactly four bits of binary, making conversions between hexadecimal and binary trivial. In textual contexts, hexadecimal values are often written with a leading `0x`. Referring to Figure C-5.3, the hexadecimal value `0xabcd` corresponds to the binary value `1010 1011 1100 1101` = 43981_{10}. Similarly, `0x400` = 2^{10} = 1024_{10}, commonly written `1K`, and `0x100000` = 2^{20} = 1048576_{10}, commonly written `1M`.

Perhaps the most obvious representation for signed integers would reserve one bit to indicate the sign (+ or −) and use the remaining $n - 1$ bits to represent the magnitude, as in unsigned numbers. Unfortunately, this approach requires different algorithms (and hence separate circuits) for addition and subtraction. The almost universally adopted alternative is called *two's complement* arithmetic. It capitalizes on the observation that arithmetic on unsigned n-digit numbers, when we ignore carries out of the left-most place, is actually arithmetic on what mathematicians call the *ring of integers modulo* 2^n. The sum $A + B$, for example, is really $(A + B) \bmod 2^n$. There is no particular reason, however, why we need to interpret the bit patterns on which we are doing our arithmetic as the numbers $0\,..\,2^n - 1$. We can actually pick any contiguous range of 2^n integers, anywhere on the number line, and say that we're doing wrap-around arithmetic on them instead. In particular, we can pick the range $-2^{n-1} \ldots 2^{n-1} - 1$.

The smallest n-digit two's complement value, -2^{n-1}, is represented by a one followed by $n-1$ zeros. Successive values are obtained by repeatedly adding one, using ordinary place-value addition. This choice of representation has several desirable properties:

1. Non-negative numbers have the same bit patterns as they do in unsigned format.
2. The most significant bit of every negative number is one; the most significant bit of every non-negative number is zero.
3. A single addition algorithm works for all combinations of negative and non-negative numbers.

EXAMPLE 5.4

Two's complement

A list of 4-bit two's complement numbers appears in Figure C-5.4. ■

The addition algorithm for both unsigned and two's complement binary numbers is the obvious binary analogue of the familiar right-to-left addition of decimal numbers. Given a fixed word size, however we must consider the issue of *overflow*. By definition we should see overflow whenever the sum of two integers (not the bit patterns, but the actual integers they represent) is outside the range of values that can be represented in 2^n bits. For unsigned integers, this is easy: overflow occurs when we have a carry out of the most significant (left-most) place. For two's complement numbers, detection is somewhat trickier. First, note that the sum of a negative and a positive number can never overflow: the result is guaranteed to be closer to zero than the larger-magnitude addend. But if the sum is positive (it has a zero left-most bit), then there must have been a carry out of the left-most place, because one of the addends had a 1 in that place.

DESIGN & IMPLEMENTATION

5.3 How much is a megabyte?

The fact that $2^{10} \approx 10^3$ facilitates "back-of-the-envelope" approximations, but can sometimes lead to confusion when precision is required. Which meaning is intended when we see `1K` and `1M`? The answer, sadly, depends on context. Main memory sizes and addresses are typically measured with powers of two, while other quantities are measured with powers of ten. Thus a 1 GHz, 1 GB embedded computer may start a new instruction 1,000,000,000 times per second, but have 1,073,741,824 bytes of memory. Its 100 GB SSD will hold 10^{11} bytes. When precision is important, careful writers will use alternative units in which "bi" (for "binary") is substituted into the second syllable and the letter 'i' is inserted into the abbreviation. Our hypothetical machine would be said to have 1 gibibyte (GiB) of memory.

0111	7	1111	−1
0110	6	1110	−2
0101	5	1101	−3
0100	4	1100	−4
0011	3	1011	−5
0010	2	1010	−6
0001	1	1001	−7
0000	0	1000	−8

Figure 5.4 **Four-bit two's complement numbers.** Note that there is a negative number (−8) that doesn't have a positive equivalent. There is only one zero, however.

EXAMPLE 5.5

Overflow in two's complement addition

If we discard carries out of the left-most place (i.e., we stay within the ring of integers mod 2^n), then we can decree that two's complement overflow has occurred when we add two non-negative numbers and get an apparently negative result (because we wrapped past the largest positive number), or when we add two negative numbers and get an apparently non-negative result (because we wrapped past the smallest [largest magnitude] negative number). For example, with 4-bit two's complement numbers, 1100 + 0110 (−4 + 6) does not overflow, even though there is a carry out of the left-most place (which we discard). On the other hand, 0101 + 0100 (5 + 4) yields 1001, an apparently negative result for positive addends, and 1011 + 1100 (−5 + −4) yields 0111 in the low four bits, an apparently positive result for negative addends. Both of these cases indicate overflow.[2]

Different machines handle overflow in different ways. Some generate a fault (a hardware exception) on overflow. Some set a bit that can be tested in software. Some provide two `add` instructions, one for each option. Some provide a single `add` that can be made to do either, depending on the value of a bit in a special register.

It turns out that one can obtain the additive inverse of a two's complement number by flipping all the bits, adding one, and discarding any carry out of the left-most place (we defer a proof to Exercise C-5.7). Subtraction can thus be implemented almost trivially using an adder, by flipping the bits of the subtrahend, providing a one as the "carry" into the least-significant place, and "adding" as usual. Multiplication and division of signed numbers are a bit trickier than addition and subtraction, but still more or less straightforward.

Note that if we take any two's complement number and its additive inverse and add them together as if they were unsigned values, keeping the final carry bit, the sum will be 2^n. This observation is the source of the name "two's complement." Of course if we discard the carry bit we get zero, which is what one would expect of $k + (-k)$.

[2] Exercise C-5.6 considers an alternative but equivalent overflow detection mechanism, which is particularly easy to implement in hardware.

5.2.2 Floating-Point Arithmetic

Floating-point numbers are the computer equivalent of scientific notation: they consist of a *mantissa* or *significand*, *sig*, an *exponent*, *exp*, and (usually) a sign bit, *s*. The value of a (binary) floating-point number is then $-1^s \times sig \times 2^{exp}$. Prior to the mid-1980s, floating-point formats and semantics tended to vary greatly across brands and even models of computers. Different manufacturers made different choices regarding the number of bits in each field, their order, and their internal representation. They also made different choices regarding the behavior of arithmetic operators with respect to rounding, overflow, underflow,[3] invalid operations, and the representation of numbers that are almost—but not quite—too small to represent. With the completion in 1985 of IEEE standard number 754 (extended in 2008), the situation changed dramatically. Most processors developed in subsequent years conform to the formats and semantics of this standard.

The 1985 version of the IEEE 754 standard defines two sizes of floating-point numbers. *Single-precision* numbers have a sign bit, eight bits of exponent, and 23 bits of significand. They are capable of representing numbers whose magnitudes vary from roughly 10^{-38} to 10^{38}. *Double-precision* numbers have 11 bits of exponent and 52 bits of significand. They represent numbers whose magnitudes vary from roughly 10^{-308} to 10^{308}. The exponent is *biased* by subtracting the most negative possible value from it, so that it may be represented by an unsigned number. In single-precision, for example, the exponent 12 is represented by the value $12 - (-127) = 139 =$ `0x8b`. The exponent -12 is represented by the value $-12 - (-127) = 115 =$ `0x73`. ■

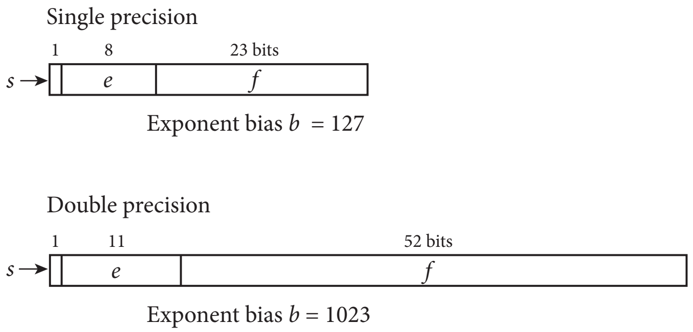
EXAMPLE 5.6
Biased exponents

Most values in the IEEE standard are *normalized* by shifting the significand until it is greater than or equal to 1, and less than 2. (The exponent is adjusted accordingly, so that the value represented doesn't change.) After normalization, we know that the leading bit of the significand will always be one, and need not be stored explicitly: to represent the value $1.\mathit{something} \times 2^{exp}$, we only need bits for the fractional part and the exponent. Exceptions to this rule occur near zero: very small numbers can be represented (with reduced precision) as $0.\mathit{something} \times 2^{min+1}$, where *min* is the smallest (most negative) exponent available in the format. Many older floating-point standards disallow such *subnormal numbers*, leading to a *gap* between zero and the smallest representable positive number that is larger than the gap between the two smallest representable positive numbers. Because it includes subnormals, the IEEE standard is said to provide for *gradual underflow*. Subnormal numbers are represented with a zero in the exponent field (denoting a maximally negative exponent) together with a nonzero fraction. (In the 1985 version of the standard, subnormal numbers were referred to as *denormal*.)

EXAMPLE 5.7
IEEE floating-point

Key conventions of the IEEE 754 standard are summarized in Figure C-5.5. In addition to the single- and double-precision formats shown here, the 2008

[3] *Underflow* occurs when the result of a computation is too close to zero to represent—that is, when its exponent is a negative number whose magnitude is too large to represent in the number of available bits.



	e	f	Value
Zero	0	0	± 0
Infinity	$2b + 1$	0	± 1
Normalized	$1 \leq e \leq 2b$	<*any*>	$\pm 1.f \times 2^{e-b}$
Denormalized	0	$\neq 0$	$\pm 0.f \times 2^{1-b}$
NaN	$2b + 1$	$\neq 0$	NaN

Figure 5.5 The IEEE 754 floating-point standard. For normalized numbers, the exponent is $e - 127$ or $e - 1023$, depending on precision. The significand is $(1 + f) \times 2^{-23}$ or $(1 + f) \times 2^{-52}$, again depending on precision. Field f is called the *fractional part*, or *fraction*. Bit patterns in which e is all ones (255 for single-precision, 2047 for double-precision) are reserved for infinities and NaNs. Bit patterns in which e is zero but f is not are used for subnormal (gradual underflow) numbers.

revision of the standard defines 16-bit *half-precision* and 128-bit *quad-precision* binary formats, as well as decimal (power-of-ten) formats in 32-, 64-, and 128-bit lengths. Both the old and new versions of the standard also permit vendor-defined "extended" formats that exceed the precision of some standard format (this accommodates, among other things, the 80-bit internal format of legacy floating point in x86 processors). We focus here on the single- and double-precision binary formats, which remain the most widely used. ■

Floating-point arithmetic is sufficiently complicated that entire books have been written about it. Some of the characteristics of the IEEE standard of particular interest to compiler writers include:

- Zero is represented by a bit pattern consisting entirely of zeros. There is also (confusingly) a "negative zero," consisting of a sign bit of one and zeros in all other positions.
- Two bit patterns are reserved to represent positive and negative infinity. These values behave in predictable ways. For example, any positive number divided by zero yields positive infinity. Similarly, the arctangent of positive infinity is $\pi/2$.
- Certain other bit patterns are reserved for special "not-a-number" (NaN) values. These values are generated by nonsensical operations, such as square root of a

negative number, addition of positive and negative infinity, or division of zero by zero. Almost any operation on an NaN produces another NaN. As a result, many algorithms can dispense with internal error checks: they can follow the steps that make sense in the absence of errors, and then check the final result to make sure it's not an NaN. Some NaNs, not normally generated by arithmetic operations, can be set by the compiler explicitly to represent uninitialized variables or other special situations; these *signaling NaNs* produce a hardware exception if used.

- The bit patterns used to represent non-negative, non-NaN floating-point numbers are ordered in the same way as integers. As a result, an ordinary integer comparison operation can (in certain contexts) be used to determine which of two numbers is larger.

An excellent introduction to both integer and floating-point arithmetic, together with suggestions for further reading, can be found in David Goldberg's appendix to Hennessy and Patterson's architecture text [HP17, App. J].

✓ CHECK YOUR UNDERSTANDING

1. Explain how to compute the additive inverse (negative) of a two's complement number.
2. Explain how to detect overflow in two's complement addition.
3. Do two's complement numbers use a bit to indicate their sign? Explain.
4. Summarize the key features of IEEE 754 floating-point arithmetic.
5. What is the approximate range of single- and double-precision floating-point values? What is the precision (in bits) of each?
6. What is a floating-point NaN?

5.3 Instruction Set Architecture (ISA)

The instructions available on a given machine, and their encoding in machine language, are referred to as the *instruction set architecture* (ISA). Existing ISAs vary quite a lot, but all include instructions for:

Computation — arithmetic and logical operations, tests, and comparisons on values held in registers (and possibly in memory)

Data movement — loads from memory to registers, stores from registers to memory, copies from one register (or memory location) to another

Control flow — conditional and unconditional branches (`goto`s), subroutine calls and returns, traps into the operating system

As we shall see in Section C-5.4, there have been several points in history at which the dominant style of instruction set design has undergone significant change. In particular, in the early to mid-1980s, designers shifted from an emphasis on *complex instruction set computing* (CISC), which sought to maximize the useful work performed per machine instruction, to *reduced instruction set computing* (RISC), which sought to maximize the number of instructions that could be completed per second. Some of the largest differences among machines today can be seen in those whose ISAs began their evolution before and after 1980. In Section C-5.4.5 we will consider one ISA in each camp: the x86, begun in 1976, and Arm, begun in 1983.

Among ISAs still in widespread use, significant differences can be seen in *addressing modes*, which specify the locations of operands; condition testing and branching; and the bit-level encoding of instructions. We will consider the first two of these in Sections C-5.3.1 and C-5.3.2 below. In the area of encoding, the most important design decision is whether to specify each instruction and its operands in a fixed, constant number of bits (typically 32), or whether to use different numbers of bits for different instructions or different numbers of arguments. Fixed-length instructions have the benefit of uniformity: they make it easier to locate and decode successive instructions, thereby facilitating the construction of *pipelined* processors (to be discussed in Section C-5.4). At the same time, certain natural operations require more than 32 bits of encoding, and thus cannot be captured in a single instruction, and certain common operations are sufficiently simple that 32 bits may constitute a waste of space. As we shall see in Section C-5.4.5, many Arm processors support an optional "Thumb mode" with shorter (16-bit) instructions. The RISC-V standard includes optional 16-bit "compressed" instructions that interoperate with standard 32-bit instructions.

From an architectural and performance perspective, the distinction between CISC and RISC ISAs is no longer of great concern: modern implementations of CISC ISAs (e.g., all recent x86 and z Series processors) incorporate a hardware "front end" that translates the legacy ISA, on the fly, into a RISC-like internal form amenable to heavily pipelined execution.

5.3.1 Addressing Modes

One can imagine many different ways in which a computational or data movement instruction might specify the location of its operand(s)—its *address*, in a broad sense of the word. A given operand might be in a register, in memory, or, in the case of read-only constants, in the instruction itself (these latter are referred to as *immediate* values).

One of the standard features of RISC machines is that computational instructions operate only on values held in registers or the instruction: a load instruction must be used to bring a value from memory into a register before it can be used as an operand. CISC machines usually allow all or most computational instructions to access operands directly in memory. RISC machines are therefore said to provide a *load-store* or *register-register* architecture; CISC machines are said to provide a *register-memory* architecture.

For binary operations, instructions on many machines can specify three addresses—two sources and a destination. Others, including the x86, provide only two-address instructions—one of the operands is always overwritten by the result. Two-address instructions are more compact, but three-address instructions are more flexible—they allow both operands to be reused in subsequent operations.

If an operand is in memory, its address might be found in a register, in memory, or in the instruction, or it might be derived from some combination of values in various locations. Instruction sets differ greatly in the *addressing modes* they provide to capture these various options. On a simple RISC machine, load and store instructions may support only the *displacement* addressing mode, in which the operand's address is found by adding some small constant (the *displacement*) to the value found in a specified register (the *base*). The displacement is contained in the instruction. Displacement addressing with respect to the frame pointer provides an easy way to access local variables. Displacement addressing with a displacement of zero is sometimes called *register indirect* addressing.

Some ISAs, including the Power family, SPARC, and Arm, also allow load and store instructions to use an *indexed* addressing mode, in which the operand's address is found by adding the values in two registers. Indexed addressing is useful for arrays: one register (the *base*) contains the address of the array; the second (the *index*) contains the offset of the desired element.

CISC machines typically provide the richest set of addressing modes, and allow them to be used in computational instructions, as well as in loads and stores. On the x86, for example, the address of an operand can be calculated by multiplying the value in one register by a small constant, adding the value found in a second register, and then adding another small constant, all in one instruction.

5.3.2 Conditions and Branches

All instruction sets provide a *branching* mechanism to update the program counter under program control. Branches allow compilers to implement conditional statements, subroutines, and loops. Conditional branches may be controlled in several ways. On many machines they use *condition codes*. As mentioned in Section C-5.1, condition codes are usually implemented as a set of bits in a special *processor status register*. All or most of the arithmetic, logical, and data-movement instructions update the condition codes as a side effect. The exact number of bits varies from machine to machine, but three and four are common: one bit each to indicate whether the instruction produced a zero value, a negative value, and/or an overflow or carry. To implement the following test, for example,

EXAMPLE 5.8

An `if` statement in x86 assembly

```
A := B + C
if A = 0 then
    body
```

a compiler for the x86[4] might generate

```
      movl    C, %eax      ; move long-word C into register eax
      addl    B, %eax      ; add
      movl    %eax, A      ; and store
      jne     L1           ; branch (jump) if result not equal to zero
       body
L1:
```

The first three instructions all set the condition codes. The fourth (`jne`) tests the codes in the wake of the `movl` that stores to A. It branches if the codes indicate that the value was not zero.

EXAMPLE 5.9

Compare and test instructions

For cases in which the outcome of a branch depends on a value that has not just been computed or moved, most machines provide `compare` and `test` instructions. Again on the x86:

```
                       movl    A, %eax      ; move long-word A into register eax
if A ≤ B then          cmpl    B, %eax      ; compare to B
    body               jg      L1           ; branch (jump) if greater
                        body
                   L1:

if A > 0 then          testl   %eax, %eax   ; compare %eax (A) to 0
    body               jle     L2           ; branch if less than or equal
                        body
                   L2:
```

The x86 `cmpl` instruction subtracts its source operand from its destination operand and sets the condition codes according to the result; it does *not*, however, overwrite the destination operand. The `testl` instruction `and`s its two operands together and compares the result to zero. Most often, as shown here, the two operands are the same. When they are different, one is typically a *mask* value that allows the programmer or compiler to test individual bits or bits fields in the other operand.

Unfortunately, traditional condition codes make it difficult to implement important performance enhancements, in both the compiler and the hardware. Because the codes are set by almost every instruction, the compiler must avoid placing unrelated instructions between the code that evaluates a condition and the branch that relies on the outcome of that evaluation. The hardware, similarly, must preserve the codes across any unrelated instructions that are executed out of order. To address these problems, the Arm and SPARC architectures make setting of the condition codes optional on an instruction-by-instruction basis. The Power architecture

[4] Readers familiar with the x86 should be warned that this example uses the assembler syntax of the GNU compiler collection (`gcc`) and its assembler, `gas`. This syntax differs in several ways from Microsoft and Intel assembly. Most notably, it specifies operands in the opposite order. The instruction `addl B, %eax`, for example, adds the value in `B` to the value in register `%eax` and leaves the result in `%eax`: in GNU assembly the *destination* operand is listed second. In Intel and Microsoft assembly it's the other way around: `addl B, %eax` would add the value in register `%ebx` to the value in `B` and leave the result in `B`.

provides eight separate sets of condition codes; compare and branch instructions can specify the set to use. MIPS and RISC-V machines eliminate condition codes entirely; instead, they provide instructions to compare two registers and branch based on the outcome.

EXAMPLE 5.10

Conditional move

Several ISAs, including Power, SPARC, and recent generations of the x86, provide a *conditional move* instruction that copies one register into another if and only if the condition codes are appropriately set. On the x86, the code fragment C := max(A, B) might naively be translated

```
    movl    A, %ecx
    movl    B, %edx
    cmpl    %edx, %ecx  ; compare %edx (A) to %ecx (B)
    jle     L1          ; branch if less than or equal
    movl    %ecx, C     ; store A to C
    jmp     L2
L1:
    movl    %edx, C     ; store B to C
L2:
```

With a conditional move instruction it can become the following instead:

```
    movl    B, %ecx
    movl    A, %edx
    cmpl    %ecx, %edx  ; compare %edx (A) to %ecx (B)
    cmovgl  %edx, %ecx  ; move A into %ecx if greater
    movl    %ecx, C     ; store to C
```

A few ISAs, including 32-bit Arm and IA-64 (Itanium), allow almost any instruction to be marked as conditional. This more general mechanism is known as *predication*. It allows an if... then ... else construct to be translated into straight-line (branch-less) code: instructions in the then and else paths are prefixed with complementary conditions, causing one path to take effect and the other to function as a sequence of *no-ops*—instructions that have no effect. When both paths are short, it may be cheaper (at least in some processor implementations) to execute the no-ops than it would have been to execute a branch.

✓ CHECK YOUR UNDERSTANDING

7. What is the most popular instruction set architecture for desktop and server machines?

8. What is the most popular instruction set architecture for tablets and cell phones?

9. What is the difference between big-endian and little-endian addressing?

10. What is the purpose of a cache?

11. Why do many machines have more than one *level* of cache?

12. How many processor cycles does it typically take to access primary (level-1) cache? How many cycles does it typically take to access main memory?
13. What is data *alignment*? Why do many processors insist upon it?
14. List four common formats (interpretations) for bits in memory.
15. What is IEEE standard number 754? Why is it important?
16. What are the tradeoffs between two-address and three-address instruction formats?
17. Describe at least five different addressing modes. Which of these are commonly supported on RISC machines?
18. What are condition codes? Why do some architectures not provide them? What do they provide instead?

5.4 Architecture and Implementation

The typical processor implementation consists of a collection of *functional units*, one (or more) for each logically separable facet of processor activity: instruction fetch, instruction decode, operand fetch from registers, arithmetic computation, memory access, write-back of results to registers, and so on. One could imagine an implementation in which all of the work for a particular instruction is completed before work on the next instruction begins, and in fact this is how the earliest computers were constructed. The problem with this organization is that most of the functional units are idle most of the time. Modern processor implementations have a substantially more complicated organization, in which the executions of many instructions *overlap* one another in time. To generate fast code, a compiler must understand the details of this organization.

Pipelining is the most fundamental form of instruction overlap. Originally developed for supercomputers of the 1960s, it moved into single-chip processors with the RISC revolution of the 1980s. On a pipelined machine, functional units work like the stations on an assembly line, with different instructions passing through different pipeline *stages* concurrently. Pipelining appears today in even the most inexpensive personal computers, and in all but the simplest processors for the embedded market. A simple processor may have 3–6 pipeline stages. The Arm Cortex-A78 (used in many cell phones) and the Intel Core i7 (used in many laptops) have 13 and 14 stages, respectively. The “superpipelined” Intel Pentium 4E had 31.

By allowing (parts of) multiple instructions to execute in parallel, pipelining can dramatically increase the number of instructions that can be completed per second, but it is not a panacea. In particular, a pipeline will *stall* if the same functional unit is needed in two different instructions simultaneously, or if an earlier instruction has not yet produced a result by the time it is needed in a later instruction, or if the outcome of a conditional branch is not known (or guessed) by the time the next instruction needs to be fetched.

We shall see in Section C-5.5 that many stalls can be avoided by adding a little extra hardware and then choosing carefully among the various ways of translating a given construct into target code. A typical example occurs in the case of floating-point arithmetic, which tends to be much slower than integer arithmetic. Rather than stall the entire pipeline while executing a floating-point instruction, we can build a separate functional unit for floating-point math, and arrange for it to operate on a separate set of floating-point registers. In effect, this strategy leads to a *pair* of pipelines—one for integers and one for floating-point—that share their first few stages. The integer branch of the pipeline can continue to execute while the floating-point unit is busy, so long as subsequent instructions do not require the floating-point result. The need to reorder, or *schedule*, instructions so that those that conflict with or depend on one another are separated in time is one of the principal reasons why compiling for modern processors is hard.

5.4.1 Microprogramming

As technology advances, there are occasionally times when it becomes feasible to design machines in a very different way. During the 1950s and the early 1960s, the instruction set of a typical computer was implemented by soldering together large numbers of discrete components (transistors, capacitors, etc.) that performed the required operations. To build a faster computer, one generally designed new, more powerful instructions, which required extra hardware. This strategy had the unfortunate effect of requiring assembly language programmers (or compiler writers, though there weren't many of them back then) to learn a new language every time a new and better computer came along.

A fundamental breakthrough occurred in the early 1960s, when IBM hit upon the idea of *microprogramming*. Microprogramming allowed a company to provide the *same* instruction set across a whole line of computers, from inexpensive slow machines to expensive fast machines. The basic idea was to build a "microengine" in hardware that executed an interpreter program in "firmware." The interpreter in turn implemented the "machine language" of the computer—in this case, the IBM 360 instruction set. More expensive machines had fancier microengines, with more direct support for the instructions seen by the assembly-level programmer. The top-of-the-line machines had everything in hardware. In effect, the architecture of the machine became an abstract interface behind which hardware designers could hide implementation details, much as the interfaces of modules in modern programming languages allow software designers to limit the information available to users of an abstraction.

In addition to allowing the introduction of computer families, microprogramming made it comparatively easy for architects to extend the instruction set. Numerous studies were published in which researchers identified some sequence of instructions that commonly occurred together (e.g., the instructions that jump to a subroutine and update bookkeeping information in the stack), and then introduced a new instruction to perform the same function as the sequence. The new instruction was usually faster than the sequence it replaced, and almost always shorter (and code size was more important then than now).

5.4.2 Microprocessors

A second architectural breakthrough occurred in the mid-1970s, when very large scale integration (VLSI) chip technology reached the point at which a simple microprogrammed processor could be implemented entirely on one inexpensive chip. The chip boundary is important because it takes much more time and power to drive signals across macroscopic output pins than it does across intrachip connections, and because the number of pins on a chip is limited by packaging issues. With an entire processor on one chip, it became feasible to build a commercially viable personal computer. Processor architectures of this era include the MOS Technology 6502, used in the Apple II and the Commodore 64, and the Intel 8080 and Zilog Z80, used in the Radio Shack TRS-80 and various CP/M machines. Continued improvements in VLSI technology led, by the mid-1980s, to 32-bit microprogrammed microprocessors such as the Motorola 68000, used in the original Apple Macintosh, and the Intel 80386, used in the first 32-bit IBM PCs.

From an architectural standpoint, the principal impact of the microprocessor revolution was to constrain, temporarily, the number of registers and the size of operands. Where the IBM 360 (*not* a single-chip processor) operated on 32-bit data, with 16 general-purpose 32-bit registers, the Intel 8080 operated on 8-bit data, with only seven 8-bit registers and a 16-bit stack pointer. Over time, as VLSI density increased, registers and instruction sets expanded as well. Intel's 32-bit 80386 was introduced in 1985.

5.4.3 RISC

By the early 1980s, several factors converged to make possible a third architectural breakthrough. First, VLSI technology reached the point at which a pipelined 32-bit processor with a sufficiently simple instruction set could be implemented on a single chip, *without* microprogramming. Second, improvements in processor speed were beginning to outstrip improvements in memory speed, increasing the relative penalty for accessing memory, and thereby increasing the pressure to keep things in registers. Third, compiler technology had advanced to the point at which compilers could often match (and sometimes exceed) the quality of code produced by the best assembly language programmers. Taken together, these factors suggested a *reduced instruction set computer* (RISC) architecture with a fast, all-hardware implementation, a comparatively low-level instruction set, a large number of registers, and an optimizing compiler.

The advent of RISC machines ran counter to the ever-more-powerful-instructions trend in processor design, but was to a large extent consistent with established trends for supercomputers. Supercomputer instruction sets had always been relatively simple and low-level, in order to facilitate pipelining. Among other things, effective pipelining depends on having most instructions take the same, constant number of cycles to execute, and on minimizing dependences that would prevent a later instruction from starting execution before its predecessors have finished.

The most basic rule of processor performance holds that total execution time on any machine equals the number of instructions executed times the length in time of a cycle times the average number of (non-overlapped) cycles per instruction (CPI). What we might call the "CISC design philosophy" is to minimize execution time by reducing the number of instructions, letting each instruction do more work. What we might call the "RISC design philosophy" is to reduce the length of the cycle and average number of cycles between the initiations of consecutive instructions. Though once cast as design alternatives, these philosophies are not mutually exclusive: complex instructions can successfully be added to a RISC design, so long as their implementation does not compromise IPC or cycle time.

High performance processors attempt to minimize CPI by executing as many instructions as possible in parallel. One core of an IBM Power10, for example, can have over 1000 instructions simultaneously "in flight" (and each processor chip has 8 cores). Some processors have very deep pipelines, allowing the work of an instruction to be divided into very short cycles. Many are *superscalar*: they have multiple parallel pipelines, and start more than one instruction each cycle. (This requires, of course, that the compiler and/or hardware identify instructions that do not depend on one another, so that parallel execution is semantically indistinguishable from sequential execution.) To minimize artificial dependences between instructions (as, for instance, when one instruction must finish using a register as an operand before another instruction overwrites that register with a new value), many machines perform *register renaming*, dynamically assigning logically independent uses of the same architectural register to different locations in a larger set of physical (implementation) registers. A high performance processor may actually execute mutually independent instructions *out of order* when it can increase instruction-level parallelism by doing so. These techniques dramatically increase implementation complexity but not architectural complexity; in fact, it is architectural *simplicity* that makes them possible.

5.4.4 Multithreading and Multicore

For 50 years, improvements in silicon fabrication technology have fueled a seemingly inexorable increase in the density of integrated circuits. This trend, first observed by Gordon Moore in 1965, has seen the number of transistors on a chip double roughly every two years since the mid 1960s—a million-fold increase since the early 1970s. Processor designers have used this amazing windfall in several major ways:

Faster clocks. Since smaller transistors can charge and discharge more quickly, higher-density chips can run at a higher clock rate. The Intel 8080 ran at 2 MHz in 1974. Rates in excess of 2 GHz (1000× faster) are commonplace today.

Instruction-level parallelism (ILP). As noted in the previous subsection, modern processors employ pipelined, superscalar, and out-of-order execution to keep a very large number of instructions "in flight," and to execute those instructions as soon as their operands become available.

Speculation. To keep the pipeline full, a modern processor guesses which way control will go at every branch, and *speculatively* executes instructions along the predicted control path. Some processors employ additional forms of speculation as well: they may, for example, guess the value that will be returned by a read that misses in the cache. So long as guesses are right, the processor avoids "unnecessary" waiting. It must always check after the fact, however, and be prepared to undo any erroneous operations in the event that a guess was wrong.

Larger caches. As noted in Sidebar C-5.2, caches play a critical role in coping with the processor-memory gap induced by higher clock rates. Higher VLSI density makes room for larger caches.

Unfortunately, by roughly 2004, the first three of these standard techniques had pretty much hit a dead end. Both faster clocks and speculation lead to very high energy consumption. To first approximation, a chip's energy requirements are proportional to its physical area and clock frequency. While caches take less energy than average (they're comparatively passive), the bookkeeping circuits required for speculation are very power-hungry. Where the 8080 consumed about 1.3 W, a desktop processor today may consume 150 W—more heat per unit area than the burner of a hot plate, and essentially at the limit of what we can cool without refrigeration. Simultaneously, ILP exploitation and speculative execution have approached the inherent limits of traditional sequential code. Bluntly put, modern single-core processors execute as many instructions in parallel as traditional programs will allow.

Robbed of the ability to run a single program faster, processor designers began building *multithreaded* and *multicore* chips that can run more than one program at once. Multithreading was introduced first. It allows several programs (threads), represented by several sets of registers and instruction fetching logic, to share the back end (execution units) of a single processor. In effect, the extra threads serve to fill "bubbles" (stalls) in the processor's pipeline. A multicore processor, by contrast, has the equivalent of two or more complete processors (cores) on a single chip (by convention, a single chip is referred to as "a processor," regardless of the number of cores). Compared to a high-end turn-of-the-century uniprocessor (a single-core machine), the cores of a modern chip may run at a somewhat slower clock rate, and expend less energy on speculation and ILP discovery, in order to maximize performance per watt.

In moving to multicore processors, the computer industry effectively gave up on running conventional programs faster, and is banking instead on running better programs. This makes the multicore revolution very different from previous changes in design philosophy. Where previous changes were mostly invisible to programmers (code might perhaps have to be recompiled to make the best use of a new machine), the multicore revolution has required programs to be *rewritten* in explicitly concurrent languages.

Unfortunately, parallel programming is hard. In practice, programs that can make effective use of hundreds or thousands or cores tend to be highly regular, applying the same operations concurrently to elements of some very large data set. For such calculations, traditional CPU architectures are overkill. As a result, we are now in the throes of yet another revolutionary change in computer architecture—one that relies on massively parallel hardware *accelerators* for big-data calculations. This revolution began with the development of general-purpose graphical processing units (GPUs), and has continued with accelerators for image processing, media transcoding, encryption, compression, and neural network training and execution. Efforts to use accelerators more frequently and effectively will be a major focus of systems research over the coming decade, but we can already see the point of diminishing returns. What will come next is currently very unclear.

✓ CHECK YOUR UNDERSTANDING

19. What is microprogramming? What breakthroughs did its invention make possible?

20. What technological threshold was crossed in the mid-1970s, enabling the introduction of microprocessors? What subsequent threshold, crossed in the early 1980s, made RISC machines possible?

21. What is pipelining?

22. Summarize the difference between the CISC and RISC philosophies in instruction set design.

23. Why do RISC machines allow only load and store instructions to access memory?

24. Name three CISC architectures. Name three RISC architectures. (If you're stumped, see the Summary and Concluding Remarks [Section C-5.6].)

25. How can the designer of a pipelined machine cope with instructions (e.g., floating-point arithmetic) that take much longer than others to compute?

26. Why are microprocessor clock rates no longer increasing?

27. Explain the difference between *multithreaded* and *multicore* processors.

28. How does the multicore revolution differ from major previous changes in computer architecture? What special problems does it pose?

5.4.5 Two Example Architectures: The x86 and Arm

EXAMPLE 5.11

The x86 ISA

We can illustrate much of the variety in ISA design—including the CISC and RISC philosophies—by examining a pair of representative architectures. The x86 is the most widely used ISA in the server, desktop, and laptop markets. The original implementation, the 8086, was announced in 1978. Major changes were introduced in Intel's 8087, 80286, 80386, Pentium Pro, Pentium/MMX, Pentium III, and Pentium 4, and in AMD's K8 (Opteron). Though technically backward compatible, these changes were often out of keeping with the philosophy of earlier generations. The result is an architecture with numerous stylistic inconsistencies and special cases. While both AMD and Intel have trade names for the instruction set, the name "x86" is widely used to refer to it generically. When necessary, "x86-32" and "x86-64" are used to refer to the 32- and 64-bit versions, both of which are in widespread use today. Some vendors use "x64" to refer to the 64-bit version.

Early generations of the x86 were extensively microprogrammed. More recent generations still use microprogramming for the more complex portions of the instruction set, but simpler instructions are translated directly (in hardware) into between one and four microinstructions that are in turn fed to a heavily pipelined, RISC-like computational core.

EXAMPLE 5.12

The Arm ISA

The original version of the Arm architecture, developed by Acorn Computers of Cambridge, England, was announced in 1983. Acorn's contemporary descendant, Arm Holdings, oversees the evolution of the instruction set, and designs—but does not fabricate—implementations. The company licenses both the instruction set and the designs to scores of partner firms, which incorporate Arm processors in everything from toasters and fuel injectors to cell phones and tablet computers—and, increasingly, to desktops and servers as well.

Like the x86, Arm has evolved considerably over time, and given its wide range of applications, it is available in a bewildering array of versions; these vary not only in speed and cost, but also in feature set. Unlike the x86, Arm never had 8- or 16-bit versions: until recently it was 32-bit only. A 64-bit extension, Arm v8, designed to compete in the desktop and server markets, was announced in 2011; the first commercial implementations became available in 2013.

Among the most significant differences between the x86 and Arm are their memory access mechanisms, their register sets, and the variety of instructions they provide. Like all RISC architectures, Arm allows only load and store instructions to access memory; all computation is done with values in registers (or in immediate fields of the current instruction). Like most CISC architectures, the x86 allows computational instructions to operate on values in either registers or memory. Like most RISC architectures, 64-bit Arm has 32 integer registers and 32 floating-point registers. On 32-bit Arm machines, there are only 16 integer registers (and only 16 are visible on a 64-bit machine when running in 32-bit mode). The x86, by contrast, has 16 registers of each kind when running in 64-bit mode, and only 8 in 32-bit mode. (There is also a separate set of 8 floating-point registers, 80 bits in length. These are used by an older set of floating-point instructions; they are increasingly ignored by modern compilers.) Arm provides many fewer distinct instructions than does the x86, and its instruction set is much more internally consistent; the x86 has a huge number of special cases. Arm instructions are normally 4 bytes long, though there is a special version of 32-bit mode called "Thumb" that provides 2-byte encodings of the most commonly used instructions. Instructions on the x86 vary from 1 to 15 bytes in length.

Memory Access and Addressing Modes

Although Arm is a register-register architecture, while the x86 is register-memory, the addressing modes of the two machines are actually quite similar: Arm has a richer set of options than many other RISC designs.

In 32-bit mode, an Arm address is formed by adding an *offset* to the value in a specified *base* register. The offset can be either an immediate *displacement* or the value in a second, *index* register. In the latter case, the offset can be shifted (*scaled*) up to 31 bit positions, effectively multiplying it by an arbitrary power of 2. With either kind of offset, the base register can optionally be updated (either before or after using its value), by adding or subtracting the (already shifted) offset. This *pre-* or *post-indexing* mechanism facilitates iteration through arrays. To economize on encoding bits, some of the addressing combinations are unavailable in 64-bit mode.

As we shall see under "Registers" below, 32-bit Arm assigns a register number to the program counter (PC), allowing that register to be used at the base in load and store instructions. This convention makes it easy to read values from the code of the running program—a trick that facilitates the construction of *position-independent code* (to be discussed in Section C-15.7.1). It also means that a branch is simply a write to the PC.

On the x86, an address is also formed by adding an offset to the value in a base register, but in this case the offset can reflect *both* an immediate displacement *and* the (possibly scaled) value in an index register. Scaling factors are less general than on Arm: possible values are 1, 2, 4, and 8. Pre- and post-increment options are also unavailable, though there are separate push and pop instructions that use the stack pointer (SP) as a base register, and automatically update it. A special PC-relative addressing mode is available in 64-bit mode, but not in 32-bit mode.

X86 instructions are two-address: the result of a computation overwrites one of the operands, which may be in either a register or memory. Computation is normally three-address on Arm (two sources and a destination can all be separate registers), but two-address when running in Thumb mode.

Registers

EXAMPLE 5.13
x86 and Arm register sets

The user-visible registers of the two architectures are illustrated pictorially in Figure C-5.6. As is immediately obvious, the Arm registers are both more numerous and more regular in structure than those of the x86. To a large extent this reflects the designs' respective histories. The 8086 was introduced in 1978 with 16-bit integer registers. (It was source-code compatible, though not binary compatible, with the earlier 8-bit 8080.) Intel expanded the registers to 32 bits in 1985 with the 80386, and AMD expanded them again to 64 bits in 2000. Arm, by contrast, has

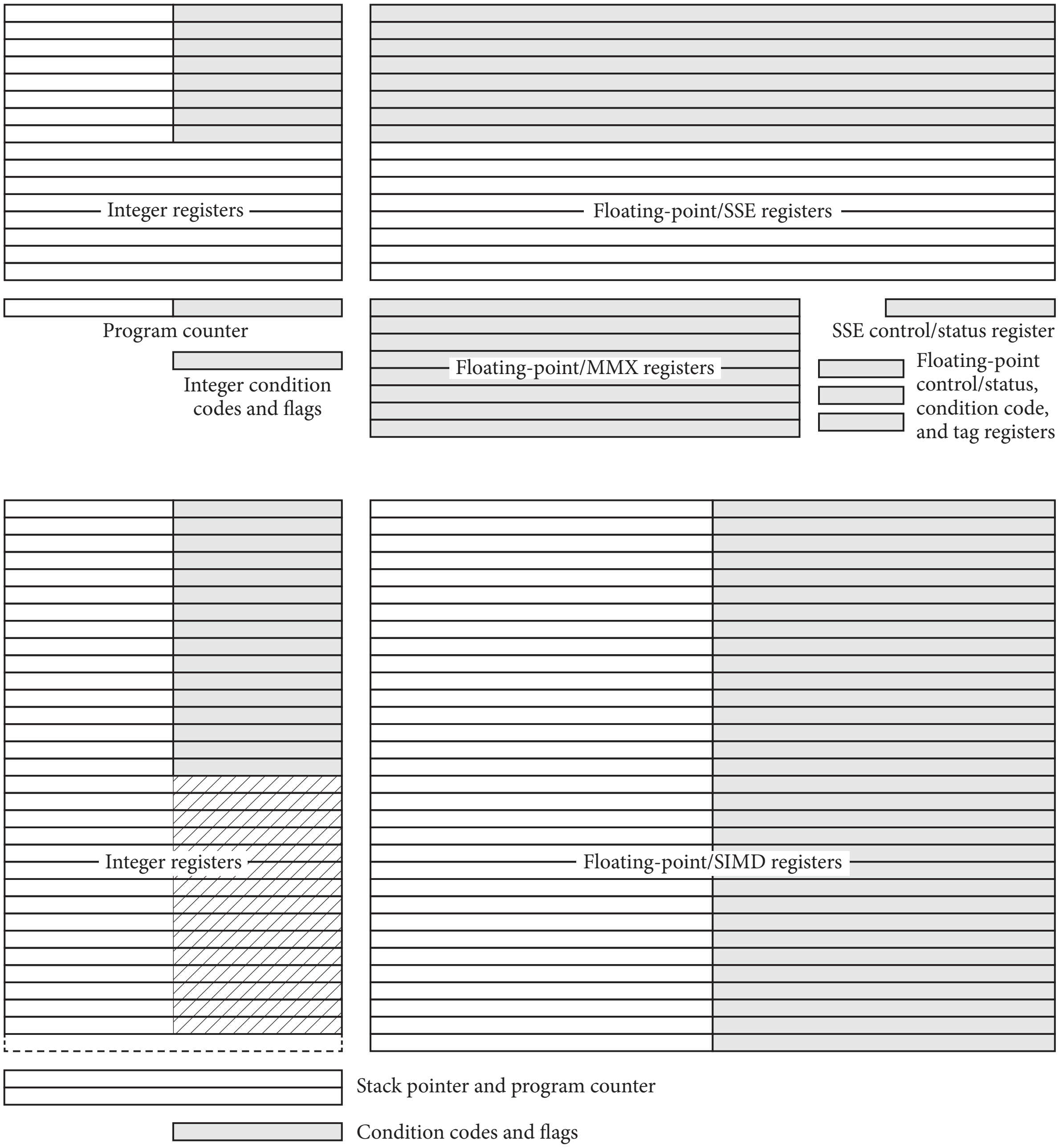

Integer registers
Floating-point/SSE registers
Program counter
SSE control/status register
Floating-point/MMX registers
Integer condition codes and flags
Floating-point control/status, condition code, and tag registers
Integer registers
Floating-point/SIMD registers
Stack pointer and program counter
Condition codes and flags

Figure 5.6 User-visible registers of the x86-64 (top) and Arm v8 (bottom). For both architectures, shaded areas indicate the subset visible in 32-bit mode. The last of the integer registers on Arm (shown with a dotted line) is virtual; it behaves as if it always contained a zero. The cross-hatched area indicates "banked" copies of the 32-bit registers, which are mapped into the bottom halves of the higher-numbered 64-bit registers. Other special registers, of use only in privileged code, are omitted for both architectures. Also omitted are the AVX registers of recent high-end x86 processors and the eight *segment registers* of the x86, which support the obsolete 80286 addressing system, and are not (for the most part) employed by modern compilers.

seen less re-engineering. It was introduced with 32-bit registers in 1983, and was extended once, to 64 in 2011.

The x86-32 has eight 32-bit integer registers, plus the program counter and the processor status word, which includes the condition codes. For historical reasons, the integer registers are named `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `esp`, and `ebp`. They can be used interchangeably in most instructions, but certain instructions use them in special ways. Registers `eax` and `edx`, for example, are implicitly the destination registers for integer multiplication and division operations. Register `ecx` is read and updated implicitly by certain loop-control instructions. Registers `esi` and `edi` are used implicitly by instructions that copy, search, or compare strings of characters in memory. Register `esp` is used as a stack pointer; it is read and written implicitly by `push`, `pop`, and subroutine call/return instructions. Register `ebp` is typically used as a frame pointer; it is manipulated by instructions designed to allocate and deallocate stack frames.

For backward compatibility with 16-bit code, there are separate names for the lower halves of all eight integer registers: `ax`, `bx`, `cx`, `dx`, `si`, `di`, `sp`, and `bp`. Four of these (`ax`, `bx`, `ax`, and `ax`) have separate names for their upper and lower halves: `ah`, `al`, `bh`, `bl`, `ch`, `cl`, `dh`, and `dl`. The x86-64 doubles the length of the 32-bit registers, naming them `rax`, `rbx`, `rcx`, `rdx`, `rsi`, `rdi`, `rsp`, and `rbp`. It then adds another 8, named `r8` through `r15`. Register `rbp` is no longer used as a frame pointer in 64-bit mode.

Floating-point instructions were originally designed (in the 8087) to operate on a stack of eight additional registers, each 80 bits in length. Three 16-bit companion registers hold IEEE floating-point status and control, floating-point condition codes, and "tag" bits that indicate whether the values in the various floating-point registers are normal, subnormal, NaN, or garbage. All computation in this legacy "x87" portion of the instruction set is performed in extended precision; values are converted to and from IEEE single- and double-precision floating-point when written to or read from memory.

Vector instructions were added to the x86 with the Pentium/MMX in 1997. To avoid requiring the operating system to save additional state when switching between processes, MMX instructions were designed to share the x87 registers. In practice the arrangement proved less than ideal: the extra internal precision of x87 floating point could cause programs to behave differently than they did on other IEEE 754-compliant machines, and stack-based addressing impeded code improvement. Moreover MMX lacked support for floating-point vectors, and the small total number of registers made it difficult to use vectors and floating point in the same program. To a large degree, both x87 floating point and MMX have been supplanted by a series of extensions known as SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions), begun in 1999. These extensions employ a separate set of 128, 256, or 512–bit registers—8 of them in 32-bit mode, 16 in 64-bit mode—and provide full support for IEEE floating point. While some 32-bit compilers continue to use the older instructions and register file, 64-bit compilers typically use only SSE and AVX.

Arm v7 has a total of 48 registers: 16 integer and 32 floating-point, named r0–r15 and d0–d31. Registers r13, r14, and r15 double as the stack pointer (SP), link register (return address—LR), and program counter (PC), respectively. All of the integer registers are 32 bits wide. There is also a 32-bit processor status register that includes the condition codes.

To facilitate fast, low-power interrupt handling in embedded applications, with minimal saving and restoring of state, Arm provides separate "banked" copies of the SP and LR register for each of several different interrupt (privilege) levels. A so-called "fast interrupt" level has additional copies of r8–r12. While these banked copies are generally of interest only to systems software, they need to be mentioned in order to fully understand the 64-bit version of the ISA.

For Arm v8, designers increased the number of integer registers to 31, doubled their width, and named them x0–x30. In a convention common to RISC machines, a 32nd "virtual register" behaves as if it always contained a zero. As shown in Figure C-5.6, the lower halves of x0–x15 overlap r0–r15. In x16–x30, the designers took the opportunity to overlap the banked copies of the 32-bit registers. This convention allows high-privilege-level 64-bit code (e.g., a virtual machine monitor) to more easily manipulate the state of medium-privilege-level 32-bit code (e.g., a guest operating system). To avoid security leaks, 64-bit code is never permitted to run at a lower privilege level than 32-bit code. The floating-point registers, for their part, were simply doubled in length, from 64 to 128 bits each. As in x86 SSE, they double as vector registers.

Register Conventions Beyond the special treatment given some registers in hardware, the designers of both the x86 and Arm recommend additional conventions to be enforced by software. On x86-32, register ebp is generally used for a frame pointer, whether or not the compiler makes use of special frame management instructions. Function values are returned in register eax (or in the pair eax:edx in the case of 64-bit return values). Any subroutine that modifies registers ebx, esi, or edi must save their old values in memory, and restore them before returning. Any caller that needs the values in eax, ecx, or edx must save them before making a call.

Additional conventions apply on x86-64. There is generally no frame pointer—rsp is used as the base when accessing data in the stack, and rbp is just an ordinary register. Moreover, the first six integer arguments to a subroutine are passed in registers rdi, rsi, rdx, rcx, r8, and r9, respectively. If there are fewer arguments, these registers must be saved by the caller if their contents are needed later. Registers rbx, rbp, r13, r14, and r15 must be saved by the callee. (Calling sequences will be discussed in more detail in Section 9.2.)

Conventions on Arm are similar. In 32-bit mode, in addition to r13, r14, and r15 (SP, LR, and PC), which are special-cased in hardware, registers r0–r3 are used by convention to hold the first four subroutine arguments and the return value, if any. Register r9 is reserved for "platform-specific" purposes; r12 is used as a scratch register for complex calls involving dynamic linking (to be discussed in Section C-15.7). In 64-bit mode, x0–x7 are used for arguments and returns, r18 is platform-specific, and `r16` and `r17` are call-time scratch registers. In both modes, registers without special purposes are divided roughly 50-50 into caller-saves and callee-saves groups.

Instructions

While it can be difficult to count the instructions in a given instruction set (the x86 can branch on any of 16 different combinations of the condition codes; does this mean it has 16 conditional branch instructions, or one with 16 variants?), it is still clear that the x86 has more, and more complex, instructions than does Arm. Some of the features of the x86 not found on Arm include:

- Binary-coded decimal arithmetic (see Sidebar 7.4).
- Character-string search, compare, and copy operations.
- Bit string search and copy operations.
- Miscellaneous "combination" instructions. These perform the same task as some multi-instruction sequence, but require less code space and presumably run faster. Examples include subroutine calls and returns, stack operations, and loop control.
- Instructions to support the obsolete 80286 segmented memory system.

On the other hand, Arm provides:

- "Building-block" instructions that perform part of some operation too complex to propagate through the pipeline as a single instruction.
- "Saturating" arithmetic, which "holds" at the extreme values of a given integer type on overflow, rather than "rolling around" mod 2^{wordsize}.
- Combination shift-and-Φ instructions, for most arithmetic operations Φ.
- Predication.
- Pre- and post-decrement addressing.

More important than any difference in the number or types of instructions, however, is the difference in how those instructions are encoded. Like most CISC machines, the x86 places a heavy premium on minimizing code size (and thus the need for memory at run time), at the expense of comparatively difficult instruction decoding. Instructions range from 1 to 15 bytes in length, with a multitude of internal formats. Similar fields do not necessarily have the same length, or appear at the same offset, in different instructions. Operand specifiers vary in length depending on the choice of addressing mode. In 64-bit (16-register) mode, the 4th bit required to name a register is not contiguous with the other 3. One-byte *prefix codes* can be prepended to certain instructions to modify their behavior, causing them to repeat multiple times, access operands in a different segment of the 80286 address space, or lock the bus for atomic access to main memory.

The instruction encodings for Arm are substantially more regular, but they have their own peculiarities. In particular, where the myriad versions of the x86 share a single, common encoding, a 64-bit Arm machine supports three separate, quite different encodings, called A32, T32, and A64. (All three can be generated from a common assembly language.)

Like most RISC ISAs, A32 devotes 32 bits to every machine instruction. Its most unusual characteristic is the reservation of 4 bits in most instructions to encode predication conditions, and a 5th to indicate whether to set the condition codes. Operations that cannot be encoded in 32 bits (e.g., because they would require a 32-bit immediate value) must be expressed with multiple instructions. To load a 32-bit value into a register, for example, one might use a `MOV` instruction to load the lower half from a 16-bit intermediate value, followed by a `MOVT` (move top) instruction to load the upper half.

Uniform instruction length has the desirable property of simplifying the construction of a pipelined processor. A shortcoming is that easily encoded (e.g., single-operand) instructions contain unneeded bits. Because it can capture such instructions in a smaller number of bytes, x86 code tends to be significantly denser than equivalent A32 code. To address this relative weakness, Arm introduced the T32 instruction set, also known as "Thumb." The most commonly executed, easily encoded instructions are specified in 16 bits in Thumb. Certain other instructions are encoded in 32 bits (though not with the same encoding as in A32). Because it lacks predication and some of the less common instructions, Thumb code tends to run slightly less quickly than equivalent A32 code. It is substantially more dense, however—a property of significant value in some embedded applications, where memory space or bandwidth may be scarce. A common practice for such applications is to compile the most performance-critical code to A32 and the rest to T32. The running program can switch from one instruction set to the other simply by executing a special branch instruction.

When designing x86-64, AMD was able to accommodate longer register names and new operations by adding an extra byte to existing instruction encodings. For Arm, fixed instruction lengths made this strategy infeasible. In a manner reminiscent of the previous design of Thumb, the company instead developed an entirely new encoding for A64—one that captures most preexisting instructions, a variety of new instructions (for 64-bit computation), and an extra bit per operand to accommodate expansion of the integer register set from 16 to 32. The key to making all of this fit in 32 bits was to reclaim the 4 bits devoted to predication in A32. The resulting instruction set and encoding are reminiscent of MIPS, Power, and SPARC, and RISC-V, all of which have always had 32 integer registers each.

As noted under "Registers" above, Arm designers chose to identify the new integer registers of A64 with the "banked" register copies reserved for (privileged) exception handling code in A32. To prevent 64-bit applications from using this capability to "spy" on more privileged code, transitions between A64 and the existing A32 and T32 encodings occur only on exceptions, when they can be mediated by the operating system—user-level code cannot change into or out of 64-bit mode the way it can transition back and forth between A32 and T32.

✓ **CHECK YOUR UNDERSTANDING**

29. Describe the most general (complex) addressing modes of the x86 and Arm architectures.
30. How many integer and floating-point registers are provided by each machine in 32-bit mode? In 64-bit mode? How wide are these registers?
31. Summarize the register usage conventions of the x86 and Arm.
32. Explain the utility of A64's "virtual" 32nd integer register.
33. List at least three "complex" instructions provided by the x86 instruction set but not provided by the Arm instruction set.
34. List at least two mechanisms provided by Arm but not by the x86.
35. Describe how floating-point support in the x86 has evolved over time.
36. Summarize the most important difference in how instructions are encoded on the x86 and Arm.
37. What is the purpose of Arm's T32 (Thumb) instruction encoding?
38. Contrast the strategies adopted by AMD and Arm in extending their respective architectures from 32 to 64 bits.

5.5 Compiling for Modern Processors

Programming a modern machine by hand, in assembly language, is a tedious undertaking. Values must constantly be shuffled back and forth between registers and memory, and operations that seem simple in a high-level language often require multiple instructions. With the rise of RISC-style instruction sets and implementations, complexity that was once hidden in microcode has been exported to the compiler. Fortunately, compilers don't get bored or make careless mistakes, and can easily deal with comparatively primitive instructions. In fact, when compiling for recent implementations of the x86, compilers generally limit themselves to a small, RISC-like subset of the instruction set, which the processor can pipeline effectively. Old programs that make use of more complex instructions still run, but not as fast; they don't take full advantage of the hardware.

EXAMPLE 5.14
Performance ≠ clock rate

The real difficulty in compiling for modern processors lies not in the need to use primitive instructions, but in the need to keep the pipeline full and to make effective use of registers. Early in this century, a user who traded in, say, a Pentium III PC for one with a Pentium 4 would typically find that while old programs ran faster on the new machine, the speed improvement was nowhere near as dramatic as the difference in clock rates would have led one to expect. Improvements would generally be better if one could obtain new program versions that were compiled with the newer processor in mind. ■

5.5.1 Keeping the Pipeline Full

Four main problems may cause a pipelined processor to stall:

1. *Cache misses.* A load instruction or an instruction fetch may miss in the cache.
2. *Resource hazards.* Two concurrently executing instructions may need to use the same functional unit at the same time.
3. *Data hazards.* An instruction may need an operand that has not yet been produced by an earlier but still executing instruction.
4. *Control hazards.* Until the outcome (and target) of a branch instruction is determined, the processor does not know the location from which to fetch subsequent instructions.

All of these problems are amenable, at least in part, to both hardware and software solutions. On the hardware side, misses can generally be reduced by building larger or more highly associative caches.[5] Resource hazards, likewise, can be addressed by building multiple copies of the various functional units (though most processors don't provide enough to avoid all possible conflicts). Misses, resource hazards, and data hazards can all be addressed by *out-of-order* execution, which allows a processor (at the cost of significant design complexity, chip area, and power consumption) to consider a lengthy "window" of instructions, and make progress on any of them for which operands and hardware resources are available.

Branches constitute something like 10% of all instructions in typical programs,[6] so even a one-cycle stall on every branch could be expected to slow down execution by 9% on average. On a deeply pipelined machine one might naively expect to stall for more like five or even ten cycles while waiting for a new program counter to be computed. To avoid such intolerable delays, most high-performance processors incorporate hardware to *predict* the outcome of each branch, based on past behavior, and to execute speculatively down the predicted path, in a way that can be "rolled back" in the event of misprediction. To the extent that it predicts correctly, such a processor can avoid control hazards altogether.

On the software side, the compiler has a major role to play in keeping the pipeline full. For any given source program, there is an unbounded number of possible translations into machine code. In general we should prefer shorter translations over longer ones, but we must also consider the extent to which various translations will utilize the pipeline. On an in-order processor (one that always executes

[5] The degree of *associativity* of a cache is the number of distinct lines in the cache in which the contents of a given memory location might be found. In a one-way associative (*direct-mapped*) cache, each memory location maps to only one possible line in the cache. If the program uses two locations that map to the same line, the contents of these two locations will keep *evicting* each other, and many misses will result. More highly associative caches are slower, but suffer fewer such conflicts.

[6] This is a very rough number. For the SPEC2000 benchmarks, Hennessy and Patterson report percentages varying from 1 to 25 [HP17, 3rd ed., pp. 138–139].

instructions in the order they appear in the machine language program), a stall will inevitably occur whenever a load is followed immediately by an instruction that needs the loaded value, because even first-level cache requires at least one extra cycle to respond. A stall may also occur when the result of a slow-to-complete floating-point operation is needed too soon by another instruction, when two concurrently executing instructions need the same functional unit in the same cycle, or, on a superscalar processor, when an instruction that uses a value is executed concurrently with the instruction that produces it. In all these cases performance may improve significantly if the compiler chooses a translation in which instructions appear in a different order.

The general technique of reordering instructions at compile time so as to maximize processor performance is known as *instruction scheduling*. On an in-order processor the goal is to identify a valid order that will minimize pipeline stalls at run time. To achieve this goal the compiler requires a detailed model of the pipeline. On an out-of-order processor the goal is simply to maximize *instruction-level parallelism* (ILP): the degree to which unrelated instructions lie near one another in the instruction stream (and thus are likely to fall within the processor's instruction window). A compiler for such an out-of-order machine may be able to make do with a less detailed pipeline model. At the same time, it may need to ensure a higher degree of ILP, since out-of-order execution tends to be found on machines with several pipelines.

DESIGN & IMPLEMENTATION

5.4 Delayed branch instructions

Successful pipelining depends on knowing the address of the next instruction before the current instruction has completed, or has even been fully decoded. With fixed-size instructions a processor can infer this address easily for straight-line code, but not for the code that follows a branch. In an attempt to minimize the impact of branch delays, several early RISC machines provided *delayed branch* instructions: with these, the instruction immediately after the branch would be executed regardless of the outcome of the branch.

Unfortunately, as architects moved to more aggressive, deeply pipelined processor implementations, the number of cycles required to correctly resolve a branch became more than one could cover with a single additional instruction. A few processors were designed with an architecturally visible branch delay of more than one cycle, but this proved not to be a viable strategy: it was simply too difficult for the compiler to find enough unrelated instructions to schedule into the slots. Instead, modern processors invariably rely on a hardware *branch predictor* to guess the outcome and targets of branches early, so that the pipeline can continue execution. That said, even when hardware is able to predict the outcome of branches, it can be useful for the compiler to do so also, in order to schedule instructions to minimize load delays on the most likely cross-branch code paths.

Instruction scheduling can have a major impact on resource and data hazards. We will consider the topic of instruction scheduling in some detail in Section C-17.6. In the remainder of the current subsection we focus on the case of loads, where even an access that hits in the cache has the potential to delay subsequent instructions.

Software techniques to *reduce* the incidence of cache misses typically require large-scale restructuring of control flow or data layout. Though aggressive optimizing compilers may reorganize loops for better cache locality, especially in scientific programs (a topic we will consider in Section C-17.7.2), most simply assume that every memory access will hit in the L1 cache, and aim to *tolerate* the delay that such a hit entails. The hit assumption is generally a good one: most programs on most machines find their data in (some level of) the cache more than 90% of the time (often over 99%). The goal of the compiler is to make sure that the pipeline can continue to operate during the time that it takes the cache to respond.

Consider a load instruction that hits in the L1 cache. The number of cycles that must elapse before a subsequent instruction can use the result is known as the *load delay*. Even the fastest caches induce a one-cycle load delay. If the instruction immediately after a load attempts to use the loaded value, a one-cycle *load penalty* (a pipeline stall) will occur. Longer pipelines can have load delays of two or even three cycles.

EXAMPLE 5.15
Filling a load delay slot

To avoid load penalties (in the absence of out-of-order execution), the compiler may schedule one or more unrelated instructions into the *delay slot*(s) between a load and a subsequent use. In the following code, for example, a simple in-order pipeline might incur a one-cycle penalty between the second and third instructions:

```
r2 := r1 + r2
r3 := A          –– load
r3 := r3 + r2
```

If we swap the first two instructions, the penalty goes away:

```
r3 := A          –– load
r2 := r1 + r2
r3 := r3 + r2
```

The second instruction gives the first instruction time enough to retrieve A before it is needed in the third instruction. ■

To maintain program correctness, an instruction-scheduling algorithm must respect all *dependences* among instructions. These dependences come in three varieties:

Flow dependence (also called *true* or *read-after-write* dependence): a later instruction uses a value produced by an earlier instruction.

Anti-dependence (also called *write-after-read* dependence): a later instruction overwrites a value read by an earlier instruction.

Output dependence (also called *write-after-write* dependence): a later instruction overwrites a value written by a previous instruction.

EXAMPLE 5.16

Renaming registers for scheduling

A compiler can often eliminate anti- and output dependences by *renaming* registers. In the following, for example, anti-dependences prevent us from moving either the instruction before the load or the one after the add into the delay slot of the load:

```
r3 := r1 + 3        -- immovable ⤈
r1 := A             -- load
r2 := r1 + r2
r1 := 3             -- immovable ⤉
```

If we use a different register as the target of the load, however, then either instruction can be moved:

```
r3 := r1 + 3        -- movable ↓                 r5 := A          -- load
r5 := A             -- load                      r3 := r1 + 3
r2 := r5 + r2                       becomes      r1 := 3
r1 := 3             -- movable ↑                 r2 := r5 + r2
```

The need to rename registers in order to move instructions can increase the number of registers needed by a given stretch of code. To maximize opportunities for concurrent execution, out-of-order processor implementations may perform register renaming dynamically in hardware, as noted in Section C-5.4.3. These implementations possess more physical registers than are visible in the instruction set. As instructions are considered for execution, any that use the same architectural register for independent purposes are given separate physical copies on which to do their work. If a processor does not perform hardware register renaming, then the compiler must balance the desire to eliminate pipeline stalls against the desire to minimize the demand for registers (so that they can be used to hold loop indices, local variables, and other comparatively long-lived values).

DESIGN & IMPLEMENTATION

5.5 Delayed load instructions

In order to enforce the flow dependence between a load of a register and its subsequent use, a processor must include so-called *interlock* hardware. To minimize chip area, several of the very early RISC processors provided this hardware only in the case of cache misses. The result was an architecturally visible *delayed load* instruction similar to the delayed branches discussed in Sidebar C-5.4. The value of the register targeted by a delayed load was undefined in the immediately subsequent instruction slot. Filling the delay slot of a delayed load with an unrelated instruction was thus a matter of correctness, not just of performance. If a compiler was unable to find a suitable "real" instruction, it had to fill the delay slot with a *no-op* (nop). More recent RISC machines have abandoned delayed loads; their implementations are fully interlocked. Within processor families old binaries continue to work correctly; the (nop) instructions are simply redundant.

5.5.2 Register Allocation

A load-store architecture explicitly acknowledges that moving data between registers and memory is expensive. A store instruction costs a minimum of one cycle—more if several stores are executed in succession and the memory system can't keep up. A load instruction costs a minimum of one or two cycles (depending on whether the delay slot can be filled), and can cost scores or even hundreds of cycles in the event of a cache miss. In order to minimize the use of loads and stores, a good compiler must keep things in registers whenever possible. We saw an example in Chapter 1: the most striking difference between the "optimized" code of Example 1.2 and the naive code of Figure 1.7 is the absence in the former of most of the loads and stores.

Register allocation is typically a two-stage process. In the first stage the compiler identifies the portions of the abstract syntax tree that represent *basic blocks*: straight-line sequences of code with no branches in or out. Within each basic block it assigns a "virtual register" to each loaded or computed value. In effect, this assignment amounts to generating code under the assumption that the target machine has an unbounded number of registers. In the second stage, the compiler maps the virtual registers of an entire subroutine onto the architectural (hardware) registers, using the same architectural register when possible to hold different virtual registers at different times, and *spilling* virtual registers to memory when there aren't enough architectural registers to go around.

EXAMPLE 5.17

Register allocation for a simple loop

We will examine this two-stage process in more detail in Section C-17.8. For now, we illustrate the ideas with a simple example. Suppose we are compiling a function that computes the variance σ^2 of the contents of an n-element vector. Mathematically,

$$\sigma^2 = \frac{1}{n}\sum_i (x_i - \overline{x})^2 = \left(\frac{1}{n}\sum_i x_i^2\right) - \overline{x}^2$$

where $x_0 \ldots x_{n-1}$ are the elements of the vector, and $\overline{x} = 1/n \sum_i x_i$ is their average. In pseudocode,

```
double sum := 0
double squares := 0
for int i in 0 . . n−1
    sum +:= A[i]
    squares +:= A[i] × A[i]
double average := sum / n
return (squares / n) − (average × average)
```

After some simple code improvements and the assignment of virtual registers, the assembly language for this function on a modern machine is likely to look something like Figure C-5.7. This code uses two integer virtual registers (v1 and v2) and eight floating-point virtual registers (w1–w8). For each of these we can compute the range over which the value in the register is useful, or *live*. This range

```
 1.        v1 := &A          -- pointer to A[1]
 2.        v2 := n           -- count of elements yet to go
 3.        w1 := 0.0         -- sum
 4.        w2 := 0.0         -- squares
 5.        goto L2
 6.  L1:   w3 := *v1         -- A[i] (floating point)
 7.        w1 := w1 + w3     -- accumulate sum
 8.        w4 := w3 × w3
 9.        w2 := w2 + w4     -- accumulate squares
10.        v1 := v1 + 8      -- 8 bytes per double-word
11.        v2 := v2 − 1      -- decrement count
12.  L2:   if v2 > 0 goto L1
13.        w5 := w1 / n      -- average
14.        w6 := w2 / n      -- average of squares
15.        w7 := w5 × w5     -- square of average
16.        w8 := w6 − w7
17.        . . .             -- return value in w8
```

Figure 5.7 Pseudo-assembly code for a vector variance computation.

extends from the point at which the value is defined to the last point at which the value is used. For register w4, for example, the range is only one instruction long, from the assignment at line 8 to the use at line 9. For register v1, the range is the union of two subranges, one that extends from the assignment at line 1 to the use (and redefinition) at line 10, and another that extends from this redefinition around the loop to the same spot again.

Once we have calculated live ranges for all virtual registers we can create a mapping onto the architectural registers. We can use a single architectural register for two virtual registers only if their live ranges do not overlap. If the number of architectural registers required is larger than the number available on the machine (after reserving a few for such special values as the stack pointer), then at various points in the code we shall have to write (spill) some of the virtual registers to memory in order to make room for the others.

In our example program, the live ranges for the two integer registers overlap, so they will have to be assigned to separate architectural registers. Among the floating-point registers, w1 overlaps with w2–w4, w2 overlaps with w3–w5, w5 overlaps with w6, and w6 overlaps with w7. There are several possible mappings onto three architectural floating-point registers, one of which is shown in Figure C-5.8. ■

Interaction with Instruction Scheduling

From the point of view of execution speed, the code in Figure C-5.8 has at least two problems. First, of the seven instructions in the loop, nearly half are devoted to bookkeeping: updating the pointer into the array, decrementing the loop count, and testing the terminating condition. Second, when run on a pipelined machine with in-order execution, the code is likely to experience a very high number of stalls. Exercise C-5.21 explores a first step toward addressing the bookkeeping

```
         r1 := &A
         r2 := n
         f1 := 0.0
         f2 := 0.0
         goto L2
    L1:  f3 := *r1               −− no delay
         f1 := f1 + f3           −− 1-cycle wait for f3
         f3 := f3 × f3           −− no delay
         f2 := f2 + f3           −− 4-cycle wait for f3
         r1 := r1 + 8            −− no delay
         r2 := r2 − 1            −− no delay
    L2:  if r2 > 0 goto L1       −− no delay
         f1 := f1 / n
         f2 := f2 / n
         f1 := f1 × f1
         f1 := f2 − f1
         . . .                   −− return value in f1
```

Figure 5.8 **The vector variance example with architectural registers assigned.** Also shown in the body of the loop are the number of stalled cycles that can be expected on a simple in-order pipelined machine, assuming a one cycle penalty for loads, two cycle penalty for floating-point adds, and four cycle penalty for floating-point multiplies.

overhead. We consider the stalls below, and return to both problems in more detail in Chapter 17.

EXAMPLE 5.18

Register allocation and instruction scheduling

We noted in Section C-5.5.1 that floating-point instructions commonly employ a separate, longer pipeline. Because they take more cycles to complete, there can be a significant delay before their results are available for use in other instructions. Suppose that floating-point add and multiply instructions must be followed by two and four cycles, respectively, of unrelated computation (these are modest figures; real machines often have longer delays). Also suppose that the result of a load is not available for a modest one-cycle delay. In the context of our vector variance example, these delays imply a total of five stalled cycles in every iteration of the loop, even if the hardware successfully predicts the outcome and target of the branch at the bottom. Added to the seven instructions themselves, this implies a total of 12 cycles per loop iteration (i.e., per vector element).

By rescheduling the instructions in the loop (Figure C-5.9) we can eliminate all but one cycle of stall. This brings the total number of cycles per iteration down to only eight, a reduction of 33%. The savings comes at a cost, however: we now execute the multiply instruction before the first floating-point add, and must use an extra architectural register to hold on to the add's second argument. This effect is not unusual: instruction scheduling has a tendency to overlap the live ranges of virtual registers whose ranges were previously disjoint, leading to an increase in the number of architectural registers required.

On a machine with out-of-order execution, hardware is likely (with the assistance of register renaming) to transform the code of Figure C-5.8 into something

```
     r1 := &A
     r2 := n
     f1 := 0.0
     f2 := 0.0
     goto L2
L1:  f3 := *r1
     r1 := r1 + 8            -- no delay
     f4 := f3 × f3           -- no delay
     f1 := f1 + f3           -- no delay
     r2 := r2 − 1            -- no delay
     f2 := f2 + f4           -- 1-cycle wait for f4
L2:  if r2 > 0 goto L1       -- no delay
     f1 := f1 / n
     f2 := f2 / n
     f1 := f1 × f1
     f1 := f2 − f1
     . . .                   -- return value in f1
```

Figure 5.9 **The vector variance example after instruction scheduling.** All but one cycle of delay has been eliminated. Because we have hoisted the multiply above the first floating-point add, however, we need an extra architectural floating-point register.

akin to Figure C-5.9 automatically on the fly, at the expense of chip area and density. As of this writing, there is still considerable debate in the architecture community regarding the relative merits of static (compiler) and dynamic (hardware) scheduling.

The Impact of Subroutine Calls

The register allocation scheme outlined above depends implicitly on the compiler being able to see all of the code that will be executed over a given span of time (e.g., an invocation of a subroutine). But what if that code includes calls to other subroutines? If a subroutine were called from only one place in the program, we could allocate registers (and schedule instructions) across both the caller and the callee, effectively treating them as a single unit. Most of the time, however, a subroutine is called from many different places in a program, and the code improvements that we should like to make in the context of one caller may be different from the ones that we should like to make in the context of a different caller. For small, simple subroutines, the compiler may actually choose to expand a copy of the code at each call site, despite the resulting increase in code size. This *inlining* of subroutines can be an important form of code improvement, particularly for object-oriented languages, which tend to have very large numbers of very small subroutines.

When inlining is not an option, most compilers treat each subroutine as an independent unit. When a body of code for which we are attempting to perform register allocation makes a call to a subroutine, there are several issues to consider:

- Parameters must generally be passed. Ideally, we should like to pass them in registers.
- Any registers that the callee will use internally, but which contain useful values in the caller, must be spilled to memory and then reread when the callee returns.
- Any variables that the callee might load from memory, but which have been kept in a register in the caller, must be written back to memory before the call, so that the callee will see the current value.
- Any variables to which the callee might store a value in memory, but which have been kept in a register in the caller, must be reread from memory when the callee returns, so that the caller will see the current value.

If the caller does not know exactly what the callee might do (this is often the case—the callee might not have been compiled yet), then the compiler must make conservative assumptions. In particular, it must assume that the callee reads and writes *every* variable visible in its scope. The caller must write any such variable back to memory prior to the call, if its current value is (only) in a register. If it needs the value of such a variable after the call, it must reread it from memory.

With perfect knowledge of both the caller and the callee, we could arrange across subroutine calls to save and restore precisely those registers that are both in use in the caller and needed (for internal purposes) in the callee. Without this knowledge, we can choose either for the caller to save and restore the registers it is using, before and after the call, or for the callee to save and restore the registers it needs internally, at the top and bottom of the subroutine. In practice it is conventional to choose the latter alternative for at least some static subset of the register set, for two reasons. First, while a subroutine may be called from many locations, there is only one copy of the subroutine itself. Saving and restoring registers in the callee, rather than the caller, can save substantially on code size. Second, because many subroutines (particularly those that are called most frequently) are very small and simple, the set of registers used in the callee tends, on average, to be smaller than the set in use in the caller. We will look at subroutine calling sequences and inlining in more detail in Sections 9.2 and 9.2.4, respectively.

DESIGN & IMPLEMENTATION

5.6 In-line subroutines

Subroutine inlining presents, to a large extent, a classic time-space tradeoff. Inlining one instance of a subroutine replaces a relatively short calling sequence with a subroutine body that is typically significantly longer. In return, it avoids the execution overhead of the calling sequence, enables the compiler to perform code improvement across the call without performing interprocedural analysis, and typically improves locality, especially in the L1 instruction cache.

✓ **CHECK YOUR UNDERSTANDING**

39. List the four principal causes of pipeline stalls.
40. What is a pipeline *interlock*?
41. What is a *delayed branch* instruction? A *delayed load* instruction?
42. What is *instruction scheduling*? Why is it important on modern machines?
43. What is the impact of out-of-order execution on compile-time instruction scheduling?
44. What is *branch prediction*? Why is it important?
45. Describe the interaction between instruction scheduling and register allocation.
46. What is the *live range* of a register?
47. What is *subroutine inlining*? What benefits does it provide? When is it possible? What is its cost?
48. Summarize the impact of subroutine calls on register allocation.

5.6 Summary and Concluding Remarks

Computer architecture has a major impact on the sort of code that a compiler must generate, and the sorts of code improvements it must effect in order to obtain good performance. Since the early 1980s, the trend in processor design has been to equip the compiler with more and more knowledge of the low-level details of processor implementation, so that the generated code can use the implementation to its fullest. This trend has blurred the traditional dividing line between processor architecture and implementation: while a compiler can generate correct code based on an understanding of the architecture alone, it cannot generate fast code unless it understands the implementation as well. In effect, timing issues that were once hidden in the microcode of microprogrammed processors (and which made microprogramming an extremely difficult and arcane craft) have been exported into the compiler.

In the first several sections of this chapter we surveyed the organization of memory and the representation of data (including integer and floating-point arithmetic), the variety of typical assembly language instructions, and the evolution of modern architectures and implementations. As examples we compared the x86 and Arm. In the final section we discussed why compiling for modern machines is hard. The principal tasks include *instruction scheduling*, to accommodate load and branch delays and multiple functional units, and *register allocation*, to minimize memory traffic. We noted that there is often a tension between these tasks, and that both are made more difficult by frequent subroutine calls.

The past two decades have seen a shake-up in RISC machines. IBM continues to invest in Power for the server market, but its PowerPC consumer line has faded away. MIPS Technologies announced in 2021 that it was transitioning development to the RISC-V ISA. Fujitsu, the last remaining manufacturer of SPARC processors, has announced plans to phase out production in the late 2020s. Arm has been the big winner, with processors designed and sold by Apple, Motorola, nVidia, Qualcomm, Texas Instruments, and scores of others. The RISC-V open standard, originally developed at UC-Berkeley, also appears to be on the rise, with adoptions by a variety of commercial vendors; it will be interesting to watch its development over the coming years.

Despite the burden of backward compatibility, the x86 overwhelmingly dominates the desktop and server market, thanks to the marketing prowess of IBM, Intel, and Microsoft, and to the engineering prowess of Intel and AMD, which have successfully decoupled the architecture from the implementation. IBM's z architecture, for its part, enjoys a virtual monopoly in mainframe computing. While modern implementations of the x86 and z continue to implement their full respective ISAs, they do so on top of pipelined implementations with uncompromised performance.

With growing demand for a 64-bit address space, a major battle developed in the x86 world around the turn of the century. Intel undertook to design an entirely new (and very different) instruction set for their IA-64/Itanium line of processors. They provided an x86 compatibility mode, but it was implemented in a separate portion of the processor—essentially a Pentium subprocessor embedded in the corner of the chip. Application writers who wanted speed and address space enhancements were expected to migrate to the new instruction set. AMD took a more conservative approach, at least from a marketing perspective, and developed a backward-compatible 64-bit extension to the x86 instruction set; its AMD64 processors provided a much smoother upward migration path. In response to market demand, Intel subsequently licensed the AMD64 architecture (which it now calls Intel 64) for use in its 64-bit x86 processors. In designing its 64-bit extension, Arm has taken an intermediate approach: its 32- and 64-bit modes share registers, and have essentially the same instructions, but use different instruction encodings.

As Arm pushes for a growing slice of the laptop/desktop/server market, it will come into increasingly direct competition with the x86, likely resulting in ever more diverse implementations and instruction set extensions. In the development of extensions, both the CISC and RISC "design philosophies" are still very much alive [SW94]. The "CISC-ish" philosophy suggests that newly available resources (e.g., increases in chip area) be used to implement functions that would otherwise have to occur in software, such as decimal arithmetic, security, virtualization, or transactional synchronization (to be discussed in Section 13.4.5). The "RISC-ish" philosophy suggests that resources be used to improve the speed of existing functions, for example by increasing cache size, employing faster but larger functional units, increasing the number of cores, or deepening the pipeline and decreasing cycle time. Depending on one's point of view, "data-parallel" accelerators for graphics, compression, encryption, transcoding, deep learning, and the like may be consistent with either philosophy.

Heat dissipation and limited ILP are increasingly the main constraints on single-core performance. In response, all the major vendors have developed multicore versions of their respective architectures. It seems increasingly likely that future processors will be highly heterogeneous, with multiple implementation strategies—and even multiple instruction set architectures—deployed in different cores, each optimized for a different sort of program. Such processors will certainly require new compiler techniques. At perhaps no time in the past 30 years has the future of microarchitecture been in so much flux. However it all turns out, it is clear that processor and compiler technology will continue to evolve together.

5.7 Exercises

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

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

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

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

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

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

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

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

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

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

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

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

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

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

5.10 Modern compilers often find they don't have enough registers to hold all the things they'd like to hold. At the same time, VLSI technology has reached the point at which there is room on a chip to hold many more registers than are found in the typical ISA. Why are we still using instruction sets with only 32 integer registers? Why don't we make, say, 64 or 128 of them visible to the programmer?

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

5.12 Why do you think RISC machines standardized on 32-bit instructions? Why not some smaller or larger length? Aside from Arm and RISC-V, why not multiple lengths?

5.13 Consider a machine with three condition codes, N, Z, and O. N indicates whether the most recent arithmetic operation produced a negative result. Z indicates whether it produced a zero result. O indicates whether it produced a result that cannot be represented in the available precision for the numbers being manipulated (i.e., outside the range $0 \mathinner{..} 2^n$ for unsigned arithmetic, $-2^{n-1} \mathinner{..} 2^{n-1}-1$ for signed arithmetic). Suppose we wish to branch on condition `A op B`, where `A` and `B` are unsigned binary numbers, for `op` $\in \{<, \leq, =, \neq, >, \geq\}$. Suppose we subtract `B` from `A`, using two's complement arithmetic. For each of the six conditions, indicate the logical combination of condition-code bits that should be used to trigger the branch. Repeat the exercise on the assumption that `A` and `B` are signed, two's complement numbers.

5.14 We implied in Section C-5.4.1 that if one adds a new instruction to a non-pipelined, microcoded machine, the time required to execute that instruction is (to first approximation) independent of the time required to execute all other instructions. Why is it not strictly independent? What factors could cause overall execution to become slower when a new instruction is introduced?

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

5.16 Consider the following code fragment in pseudo-assembly notation:

```
 1.     r1 := K
 2.     r4 := &A
 3.     r6 := &B
 4.     r2 := r1 × 4
 5.     r3 := r4 + r2
 6.     r3 := *r3             –– load (register indirect)
 7.     r5 := *(r3 + 12)      –– load (displacement)
 8.     r3 := r6 + r2
 9.     r3 := *r3             –– load (register indirect)
10.     r7 := *(r3 + 12)      –– load (displacement)
11.     r3 := r5 + r7
12.     S := r3               –– store
```

(a) Give a plausible explanation for this code (what might the corresponding source code be doing?).

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

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

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

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

5.18 Some processors, including the Power series and certain members of the x86 family, require one or more cycles to elapse between a condition-determining instruction and a branch instruction that uses that condition. What options does a scheduler have for filling such delays?

5.19 Branch prediction can be performed statically (in the compiler) or dynamically (in hardware). In the static approach, the compiler guesses which way the branch will usually go, encodes this guess in the instruction, and schedules instructions for the expected path. In the dynamic approach, the hardware keeps track of the outcome of recent branches, notices branches or patterns of branches that recur, and predicts that the patterns will continue in the future. Discuss the tradeoffs between these two approaches. What are their comparative advantages and disadvantages?

5.20 Consider a machine with a three-cycle penalty for incorrectly predicted branches and a zero-cycle penalty for correctly predicted branches. Suppose that in a typical program 20% of the instructions are conditional branches, which the compiler or hardware manages to predict correctly 75% of the time. What is the impact of incorrect predictions on the average number of cycles per instruction? Suppose the accuracy of branch prediction can be increased to 90%. What is the impact on cycles per instruction?

Suppose that the number of cycles per instruction would be 1.5 with perfect branch prediction. What is the percentage slowdown caused by mispredicted branches? Now suppose that we have a superscalar processor on which the number of cycles per instruction would be 0.6 with perfect branch prediction. Now what is the percentage slowdown caused by mispredicted branches? What do your answers tell you about the importance of branch prediction on superscalar machines?

5.21 Consider the code in Figure C-5.9. In an attempt to eliminate the remaining delay, and reduce the overhead of the bookkeeping (loop control) instructions, one might consider *unrolling* the loop: creating a new loop in which each iteration performs the work of k iterations of the original loop. Show the code for $k = 2$. You may assume that n is even, and that your target machine supports displacement addressing. Schedule instructions as tightly as you can. How many cycles does your loop consume per vector element?

5.8 Explorations

5.22 Skip ahead to Sidebar 7.4 (Decimal types) in the main text. Write algorithms to convert BCD numbers to binary, and vice versa. Try writing the routines in assembly language for your favorite machine (if your machine has special instructions for this purpose, pretend you're not allowed to use them). How many cycles are required for the conversion?

5.23 Is microprogramming an idea that has outlived its usefulness, or are there application domains for which it still makes sense to build a microprogrammed machine? Defend your answer.

5.24 If you have access to both CISC and RISC machines, compile a few programs for both machines and compare the size of the target code. Can you generalize about the "space penalty" of RISC code?

5.25 The Intel IA-64 (Itanium) architecture is neither CISC nor RISC. It belongs to an architectural family known as *long instruction word* (LIW) machines (Intel calls it *explicitly parallel instruction set computing* [EPIC]). Find an Itanium manual or tutorial and learn about the instruction set. Compare and contrast it with the x86 and Arm instruction sets. Discuss, from a compiler writer's point of view, the challenges and opportunities presented by the IA-64.

5.26 Research the history of the x86. Learn how it has been extended over the years. Write a brief paper describing the extensions. Identify the portions of the instruction set that are still useful today (i.e., are targeted by modern compilers), and the portions that are maintained solely for the sake of backward compatibility.

5.27 If you have access to computers with more than one kind of processor, compile a few programs on each machine and time their execution. (If possible, use the same compiler [e.g., `gcc`] and options on each machine.) Discuss the factors that may contribute to different run times. How closely do the ratios of run times mirror the ratios of clock rates? Why don't they mirror them exactly?

5.28 Branch prediction can be characterized as *control speculation*: it makes a guess about the future control flow of the program that saves enough time when it's right to outweigh the cost of cleanup when it's wrong. Some researchers have proposed the complementary notion of *value speculation*, in which the processor would predict the value to be returned by a cache miss, and proceed on the basis of that guess. What do you think of this idea? How might you evaluate its potential?

5.29 Can speculation be useful in software? How might you (or a compiler or other tool) be able to improve performance by making guesses that are subject to future verification, with (software) rollback when wrong? (Hint: Think about operations that require communication over slow Internet links.)

5.30 Translate the high-level pseudocode for vector variance (Example C-5.17) into your favorite programming language, and run it through your favorite compiler. Examine the resulting assembly language. Experiment with different levels of optimization (code improvement). Discuss the quality of the code produced.

5.31 Try to write a code fragment in your favorite programming language that requires so many registers that your favorite compiler is forced to spill some registers to memory (compile with a high level of optimization). How complex does your code have to be?

5.32 Experiment with small subroutines in C++ to see how much time can be saved by expanding them in-line.

5.9 Bibliographic Notes

The standard reference in computer architecture is the graduate-level text by Hennessy and Patterson [HP17]. More introductory material can be found in the undergraduate computer organization text by the same authors [PH20]. Students without previous assembly language experience may be particularly interested in the text of Bryant and O'Hallaron [BO16], which surveys computer organization from the point of view of the systems programmer, focusing in particular on the correspondence between source-level programs in C and their equivalents in x86 assembly.

The "RISC revolution" of the early 1980s was spearheaded by three separate research groups. The first to start (though last to publish [Rad82]) was the 801 group at IBM's T. J. Watson Research Center, led by John Cocke. IBM's Power and PowerPC architectures, though not direct descendants of the 801, take significant inspiration from it. The second group (and the one that coined the term "RISC") was led by David Patterson [PD80, Pat85] at UC Berkeley. The commercial SPARC architecture is a direct descendant of the Berkeley RISC II design. The third group was led by John Hennessy at Stanford [HJBG81]. The commercial MIPS architecture is a direct descendant of the Stanford design.

Much of the history of pre-1980 processor design can be found in the text by Siewiorek, Bell, and Newell [SBN82]. This classic work contains verbatim reprints of many important original papers. In the context of RISC processor design, Smith and Weiss [SW94] contrast the more "RISCy" and "CISCy" design philosophies in their comparison of implementations of the Power and Alpha architectures. Hennessy and Patterson's architecture text includes an appendix that summarizes the similarities and differences among the major commercial instruction sets [HP17, App. K]. Current manuals for all the popular commercial processors are available from their manufacturers.

An excellent treatment of computer arithmetic can be found in Goldberg's appendix to the Hennessy and Patterson architecture text [Gol17]. Additional coverage of floating point can be found in the same author's 1991 *Computing Surveys* article [Gol91]. The IEEE 754 floating-point standard was printed in *ACM SIGPLAN Notices* in 1985 [IEE87]. The texts of Muchnick [Muc97] and of Cooper and Torczon [CT11] are excellent sources of information on instruction scheduling, register allocation, subroutine optimization, and other aspects of compiling for modern machines.

6 Control Flow

6.7 Nondeterminacy

In Algol 68, the lack of ordering among expression operands was explicitly defined as an example of nondeterminacy, which the language designers called *collateral execution*. Several other built-in constructs in Algol 68 were nondeterministic as well, and an explicit *collateral statement* allowed the programmer to specify nondeterminacy in the evaluation of arbitrary expressions when desired.

Among his many contributions to the art of programming, Dijkstra [Dij75] advocated the use of nondeterminacy for selection and logically controlled loops. His *guarded command* notation has been adopted by several languages. One of these is SR, a pedagogical language of the 1980s, which we will mention again in Chapter 13. Imagine for a moment that we are writing a function to return the maximum of two integers. In C, we would probably employ a code fragment something like this:

EXAMPLE 6.89
Avoiding asymmetry with nondeterminism

```
if (a > b) max = a;
else max = b;
```

Of course, we could also write

```
if (a >= b) max = a;
else max = b;
```

These fragments differ in their behavior when a = b: the first sets `max = b`; the second sets `max = a`. As a practical matter the difference is irrelevant, since `a` and `b` are equal, but it is in some sense aesthetically unpleasant to have to make an arbitrary choice between the two. More important, the arbitrariness of the choice makes it more difficult to reason about the code formally, or to prove it is correct. In a language with guarded commands (the example here is in SR), one could write the following:

EXAMPLE 6.90
Selection with guarded commands

```
if a >= b -> max := a
[] b >= a -> max := b
fi
```

The general form of this construct is

```
if condition -> stmt_list
[] condition -> stmt_list
[] condition -> stmt_list
...
fi
```

Each of the conditions in this construct is known as a *guard*. The guard and its following statement, together, are called a *guarded command*. When control reaches an `if` statement in a language with guarded commands, a nondeterministic choice is made among the guards that evaluate to true, and the statement list following the chosen guard is executed. In SR, the final condition may optionally be `else`. If none of the conditions evaluates to true, the statement list following the `else`, if any, is executed. If there is no `else`, the `if` statement as a whole has no effect. (In Dijkstra's original proposal, there was no `else` guard option, and it was a dynamic semantic error for none of the guards to be true.) Interestingly, SR provides no separate `case` construct: the SR compiler detects when the conditions of an `if` statement test the same expression against a nonoverlapping set of compile-time constants, and generates table-lookup code as appropriate.

EXAMPLE 6.91
Looping with guarded commands

SR uses guarded commands for several purposes in addition to selection. Its logically controlled looping construct (again patterned on Dijkstra's proposal) looks very much like the `if` statement:

```
do condition -> stmt_list
[] condition -> stmt_list
[] condition -> stmt_list
...
od
```

For each iteration of the loop, a nondeterministic choice is made among the guards that evaluate to true, and the statement list following the chosen one is executed. The loop terminates when none of the guards is true (there is no `else` guard option for loops). Using this notation, we can write Euclid's greatest common divisor algorithm as follows:

```
do a > b -> a := a - b
[] b > a -> b := b - a
od
gcd := a
```

Nondeterministic Concurrency

While nondeterministic constructs have a certain appeal from an aesthetic and formal semantics point of view, their most compelling advantages arise in concurrent programs, for which they can affect correctness. Imagine, for example, that we are writing a simple dictionary program to support computer-aided design on a

EXAMPLE 6.92
Nondeterministic message receipt

```
process client:
    loop
        toss coin
        if heads, send read request to server
                  wait for response
        if tails, send write request to server
                  wait for response

process server:
    loop
            receive read request
            reply with data
        OR
            receive write request
            update data and reply
```

Figure 6.7 **Example of a concurrent program that requires nondeterminacy.** The server must be able to accept either a `read` or a `write` request, whichever is available at the moment. If it insists on receiving them in any particular order, deadlock may result.

network of personal computers. The dictionary keeps a mapping from part names to their specifications. A dictionary server process handles requests from clients on other workstations on the network. Each request may be either a read (return me the current specification for part X) or a write (define part Y as follows).[1] Clients send requests at unpredictable times. As a result, the server cannot tell at any given time whether it should try to receive a read or a write request. If it makes the wrong choice the entire system may deadlock (see Figure C-6.7). ■

EXAMPLE 6.93

Nondeterministic server in SR

Most message-based concurrent languages provide at least one mechanism to specify nondeterministic choice among potential communication partners. These mechanisms do not all look like guarded commands, but they have similar semantics. In SR, one could write our dictionary server as follows:

```
# declarations of request types:
op read_data(n : name) returns d : description
op write_data(n : name; d : description)
# local subroutines:
proc lookup ...         # find info in dictionary
proc update ...         # change info in dictionary
```

[1] This is of course an oversimplified example. Among other things, any real system of this sort would need a mechanism to lock parts in the dictionary, so that no two clients would ever end up designing new specifications for the same part concurrently.

```
# code for server:
process server
    do true ->          # loop forever
        in read_data(n) returns d -> d := lookup(n)
        [] write_data(n, d) -> update(n, d)
        ni
    od
end
```

Here `in` is a nondeterministic construct whose guards can contain communication statements. The guard `write_data(n, d)` will evaluate to true if and only if some client is attempting to send a request containing a new specification for a part. We shall see in Section C-13.5.3 that more elaborate guards can allow a server to constrain the types of requests that it is willing to receive at a given point in time, or even to "peek" inside a message to see if it is acceptable. If none of the guards of an `in` statement is true, the server waits until one is.

Choosing among Guards

EXAMPLE 6.94

Naive (unfair) implementation of nondeterminism

What happens if two or more guards evaluate to true? How does the language implementation choose among them? We have glossed over this issue so far. The most naive implementation would treat a guarded command construct like a conventional `if`... `then`... `else`:

```
server:
    loop
        if read_data request available
            . . .
        elsif write_data request available
            . . .
        else wait until some request is available
```

The problem with this implementation is that it always favors one type of request over another; if `read_data` requests are always available, `write_data` requests will never be received.

EXAMPLE 6.95

"Gotcha" in round-robin implementation of nondeterminism

A slightly more sophisticated implementation would maintain a circular list of the guards in each set of guarded commands. Each time it encounters the construct in which these commands appear, it would check guards beginning with the one after the one that succeeded last time. This technique works well in many cases, but can fail consistently in others. In the following, for example (again in SR), the guard of the first `in` statement combines a communication test with a Boolean condition:

```
process silly
var count : int := 0
    do true ->
        in A() st count % 2 = 1 -> ...
        [] B() -> ...
        [] C() -> ...
        ni
        count++
    od
```

This example is somewhat contrived, but illustrates the problem. The `st` ("such that") clause in the first guard indicates that it can be chosen only on odd iterations of the loop. Now imagine that `A`, `B`, and `C` requests are always available. If we always check guards starting with the one after the one that succeeded last time (beginning at first with the initial guard), then `B` will be chosen in the first iteration (because `count mod 2` $\neq 1$), `C` will be chosen in the second iteration (when `count` = 2), `B` will be chosen again in the third iteration (because again `count mod 2` $\neq 1$), and so forth. `A` will never be chosen. The lesson to be learned from this example is that no deterministic algorithm will provide a truly satisfactory implementation of a nondeterministic construct (see Sidebar C-6.10).

One final issue has to do with side effects. Guarded command constructs make a nondeterministic choice among the guards that evaluate to true. They do *not*, however, guarantee that all guards will be evaluated before the choice is made; the implementation is free to ignore the rest of the guards once it has chosen one that is true. A program may therefore produce unexpected or even unpredictable

DESIGN & IMPLEMENTATION

6.10 Nondeterminacy and fairness

Ideally, what we should like in a nondeterministic construct is a guarantee of *fairness*. This turns out to be trickier than one might expect: there are several plausible ways that "fair" might be defined. Certainly we should like to guarantee that no guard that is always true is always skipped. Probably, we should like to guarantee that no guard that is true infinitely often (in a hypothetical infinite sequence of iterations) is always skipped. Better, we might ask that any guard that is true infinitely often be chosen infinitely often. This stronger notion of fairness will obtain if the choice among true guards is genuinely random. Unfortunately, good pseudorandom number generators are expensive enough that we may not want to use them to choose among guards. As a result, most implementations of guarded commands are not provably fair. Many simply employ the circular list technique. Others use somewhat "more random" heuristics. Many machines, for example, provide a fast-running clock register that can be read efficiently in user-level code. A reasonable "random" choice of the guard to evaluate first can be made by interpreting this clock as an integer, and computing its remainder modulo the number of guards.

results if any of the guards have side effects. This problem is the programmer's responsibility in SR. An alternative would have been to prohibit side effects and have the compiler verify their absence.

✓ **CHECK YOUR UNDERSTANDING**

45. What is a *guarded command*?
46. Explain why nondeterminacy is particularly important for concurrent programs.
47. Give three alternative definitions of *fairness* in the context of nondeterminacy.
48. Describe three possible ways of implementing the choice among guards that evaluate to true. What are the tradeoffs among these?

6 Control Flow

6.9 Exercises

6.38 Explain why the following guarded commands in SR are *not* equivalent:

```
if a < b -> c := a              if a < b -> c := a
[] b < c -> c := b              [] b < c -> c := b
[] else -> c := d               [] true -> c := d
fi                              fi
```

6.39 The astute reader may have noticed that the final line of the code in Example C-6.91 embodies an arbitrary choice. It could just as easily have said `gcd := b`. Show how to use a guarded command to restore the symmetry of the program.

6.40 Write, in SR or pseudocode, a function that returns

(a) an arbitrary nonzero element of a given array

(b) an arbitrary permutation of a given array

In each case, write your code in such a way that if the implementation of nondeterminism were truly random, all correct answers would be equally likely.

6 Control Flow

6.10 Explorations

6.48 Learn about the `select` routine in the Unix (POSIX) library. How does it deal with the need for nondeterministic receipt from multiple communication partners? How would you use this routine to achieve the effect of the SR code in Example C-6.93?

6.49 Explain how to use threads in Java to achieve the effect of Example C-6.93.

7 Type Systems

7.3.5 Generics in C++, Java, and C#

Though templates were not officially added to C++ until 1990, when the language was almost ten years old, they were envisioned early in its evolution. C# generics, likewise, were planned from the beginning, though they actually didn't appear until the 2.0 release in 2004. By contrast, generics were deliberately omitted from the original version of Java. They were added to Java 5 (also in 2004) in response to strong demand from the user community.

C++ Templates

EXAMPLE 7.56

Generic `arbiter` class in C++

Figure C-7.6 defines a simple generic class in C++ that we have named an `arbiter`. The purpose of an `arbiter` object is to remember the "best instance" it has seen of some generic parameter class `T`. We have also defined a generic `chooser` class that provides an `operator()` method, allowing it to be called like a function. The intent is that the second generic parameter to `arbiter` should be a subclass of `chooser`, though as written the code does not enforce this. Given these definitions we might write

```
class case_sensitive : chooser<string> {
public:
    bool operator()(const string& a, const string& b) { return a < b; }
};
...
arbiter<string, case_sensitive> cs_names;       // declare new arbiter
cs_names.consider(new string("Apple"));
cs_names.consider(new string("aardvark"));
cout << *cs_names.best() << "\n";               // prints "Apple"
```

Alternatively, we might define a `case_insensitive` descendant of `chooser`, whereupon we could write

```
template<typename T>
class chooser {
public:
    virtual bool operator()(const T& a, const T& b) = 0;
};

template<typename T, typename C>
class arbiter {
    T* best_so_far;
    C comp;
public:
    arbiter() { best_so_far = nullptr; }
    void consider(T* t) {
        if (!best_so_far || comp(*t, *best_so_far)) best_so_far = t;
    }
    T* best() {
        return best_so_far;
    }
};
```

Figure 7.6 Generic arbiter in C++.

```
arbiter<string, case_insensitive> ci_names;     // declare new arbiter
ci_names.consider(new string("Apple"));
ci_names.consider(new string("aardvark"));
cout << *ci_names.best() << "\n";               // prints "aardvark"
```

Either way, the C++ compiler will create a new instance of the `arbiter` template every time we declare an object (e.g., `cs_names`) with a different set of generic arguments. Only at the point where we attempt to use such an object (e.g., by calling `consider`) will it check to see whether the arguments support all the required operations.

Because type checking is delayed until the point of use, there is nothing magic about the `chooser` class. If we neglected to define it, and then left it out of the header of `case_sensitive` (and similarly `case_insensitive`), the code would still compile and run just fine. ■

C++ templates are an extremely powerful facility. Template parameters can include not only types, but also values of ordinary (nongeneric) types, and nested template instances. Programmers can also define *specialized* templates that provide alternative implementations for certain combinations of arguments. These facilities suffice to implement recursion, giving programmers the ability, at least in principle, to compute arbitrary functions at compile time (in other words, templates are *Turing complete*). An entire branch of software engineering has grown up around so-called *template metaprogramming*, in which templates are used to persuade the C++ compiler to generate custom algorithms for special circumstances [AG05]. As a comparatively simple example, one can write a template that accepts a generic parameter `int n` and produces a sorting routine for *n*-element arrays in which all of the loops have been completely unrolled.

As described in Section 7.3.4 ("Implicit Instantiation"), C++ allows generic parameters to be *inferred* for generic functions, rather than specified explicitly. To identify the right version of a generic function (from among an arbitrary number of specializations), and to deduce the corresponding generic arguments, the compiler must perform a complicated, potentially recursive pattern-matching operation. This pattern matching is, in fact, quite similar to the type inference of ML-family languages, described in Section 7.4. It can, as noted in Sidebar 7.8, be cast as *unification*.

EXAMPLE 7.57

Template function bodies moved to a `.cc` file

Unfortunately, per-use instantiation of templates has several significant drawbacks. First, it requires that the compiler have access to the template's source code at the point in the program where instantiation occurs. In the code of Figure C-7.6, the `arbiter` class includes complete definitions of its methods. This is entirely appropriate for small, simple classes, even in a header (`.h`) file. If the code were significantly more complex, we might wish to put only the declaration of the generic class in our header file (call it `arbiter.h`), and defer the method definitions to a separate `arbiter.cc` file:

```
// arbiter.h:

template<typename T, typename C>
class arbiter {
    T* best_so_far;
    C comp;
public:
    arbiter();
    void consider(T* t);
    T* best();
};

// arbiter.cc (imagine that these methods were long and complicated):

template<class T, class C>
arbiter<T,C>::arbiter() { best_so_far = nullptr; }

template<class T, class C>
void arbiter<T,C>::consider(T* t) {
    if (!best_so_far || comp(*t, *best_so_far)) best_so_far = t;
}

template<class T, class C>
T* arbiter<T,C>::best() { return best_so_far; }
```

Compilation units that have access to the `.h` file will still compile successfully, but now machine code for the `arbiter` methods will never be instantiated, because no actual use of an `arbiter` object appears in the file (`arbiter.cc`) that contains the source code. The likely symptom will be "missing symbol" errors from the linker.

C++ provides a partial solution to this problem, in the form of *explicit instantiation*. If we anticipate the need for case-sensitive and case-insensitive `string` arbiters, we can define the appropriate `chooser` classes in `arbiter.h`, and then instantiate corresponding `arbiter` classes in `arbiter.cc`:

```
template class arbiter<string, case_sensitive>;
template class arbiter<string, case_insensitive>;
```

Of course, explicit instantiation works only if the implementor of a template's `.cc` file knows what instantiations will eventually be required. If this cannot be anticipated, the bodies will need to remain in the `.h` file, regardless of their complexity. But then a second problem arises: if the same template is instantiated with the same arguments in 20 different compilation units, the compiler will end up compiling the same code 20 times. Most modern linkers are smart enough to keep only one copy of the machine code for a repeatedly instantiated template, but we will have wasted not only the cost of repeated scanning and parsing, but of semantic analysis, optimization, and code generation as well.

EXAMPLE 7.58

`extern` templates in C++11

C++11 provides a partial solution to this second problem, in the form of `extern` template declarations. If the templated class declaration and method definitions of Example C-7.57 were included in their entirety in `arbiter.h`, and we then needed a case-sensitive `arbiter` in each of 20 `.cc` files, we could write

```
extern template class arbiter<string, case_sensitive>;
```

in all but one of the files, instructing the compiler *not* to generate machine code for that `arbiter`, but rather to assume that an appropriate implementation would be generated elsewhere (presumably in the 20th file, where the `extern` keyword would be omitted), and would thus be available at link time.

EXAMPLE 7.59

Instantiation-time errors in C++ templates

Historically, the final and perhaps the most frustrating problem with per-use instantiation was its tendency to result in inscrutable error messages. Continuing our running example, if we define

```
class foo {                                // line 40 of source
public:
    bool operator()(const string& a, const unsigned int b) {
        // wrong type for second parameter, from arbiter's point of view
        return a.length() < b;
    }
};
```

and then say

```
arbiter<string, foo> oops;
...
oops.consider(new string("Apple"));        // line 75 of source
```

the GNU C++ compiler (version 12.2) will respond with

```
best.cc: In instantiation of ‘void arbiter<T, C>::consider(T*)
[with T = std::__cxx11::basic_string<char>; C = foo]’:
best.cc:75:18:   required from here
best.cc:28:33: error: no match for call to ‘(foo)
    (std::__cxx11::basic_string<char>&, std::__cxx11::basic_string<char>&)’
   28 |         if (!best_so_far || comp(*t, *best_so_far)) best_so_far = t;
      |                                ~~~~^~~~~~~~~~~~~~~~~~
best.cc:42:10: note: candidate: ‘bool foo::operator()(const std::string&,
    unsigned int)’
   42 |     bool operator()(const string& a, const unsigned int b) {
      |          ^~~~~~~
best.cc:42:57: note:   no known conversion for argument 2
    from ‘std::__cxx11::basic_string<char>’ to ‘unsigned int’
   42 |     bool operator()(const string& a, const unsigned int b) {
      |                                      ~~~~~~~~~~~~~~~~~~~^
```

LLVM’s `clang` front end (version 14.0) is only a little more helpful:

```
best.cc:28:29: error: no matching function for call to object of type 'foo'
        if (!best_so_far || comp(*t, *best_so_far)) best_so_far = t;
                            ^~~~
best.cc:75:10: note: in instantiation of member function
'arbiter<std::string, foo>::consider' requested here
    oops.consider(new string("Apple"));          // line 75 of source
         ^
best.cc:42:10: note: candidate function not viable: no known conversion
from 'std::string' to 'const unsigned int' for 2nd argument
    bool operator()(const string& a, const unsigned int b) {
         ^
```

The problem here is fundamental; it’s not poor compiler design. Because the language requires that templates be “expanded out” before they are type checked, it is extraordinarily difficult to generate messages without reflecting that expansion.

To facilitate more helpful messages (and also to increase the expressive power of template specialization), C++20 introduced the notion of *concepts*, which allow the programmer to specify constraints on template parameters—and the compiler to check those constraints at instantiation time. Dozens of concepts and related *requirements* are defined in the standard library, and rich notation is available in which to define additional concepts.

EXAMPLE 7.60
Insisting on a `chooser`

Perhaps the simplest constraint we might add to our `arbiter` type insists that type parameter `C` be derived from `chooser<T>`:

```
template<typename T, typename C>
    requires std::derived_from<C, chooser<T>>
class arbiter { ...
```

With this change in place, `clang` says

```
best.cc:74:5: error: constraints not satisfied for class template 'arbiter'
[with T = std::string, C = foo]
    arbiter<string, foo> oops;
    ^~~~~~~~~~~~~~~~~~~~
best.cc:18:14: note: because 'std::derived_from<foo, chooser<std::string> >'
evaluated to false
    requires std::derived_from<C, chooser<T>>
             ^
```

Note that the error message is now associated with the instantiation of `arbiter` at line 74, rather than its use at line 75.

EXAMPLE 7.61
Insisting on a predicate

But this is overkill. It rules out cases in which we provide a perfectly usable comparator type `C` that isn't actually derived from `chooser<T>`. To allow more general comparators, we might write

```
template<typename T, typename C>
    requires std::predicate<C, T, T>
class arbiter { ...
```

Here `predicate<C, T, T>` requires that `C` be an invocable object that takes two parameters of type `T` and returns a value whose type is (or can be coerced to be) `bool`. With this revised constraint, error messages still occur at the point of instantiation but we are no longer limited to strict descendants of the `chooser` type.

EXAMPLE 7.62
Defining a new named concept

Unfortunately, the error message for a failed instantiation of Example C-7.61 (a message we haven't shown) now fills most of a page, diving into details of the definition of `std::predicate`. To improve this message, we can define a concept that captures exactly what we need:

```
template<typename T, typename C>
concept Compares =
    requires(C c, T a, T b) { {c(a, b)} -> std::convertible_to<bool>; };

template<typename T, typename C>
    requires Compares<T, C>
class arbiter { ...
```

Here we have defined `Compares` to insist that for any `C` object `c` and any `T` objects `a` and `b`, the expression `c(a, b)` is well formed and has a value whose type is (or can be coerced to be) `bool`. Now our error message is quite nice:

```
best.cc:74:5: error: constraints not satisfied for class template 'arbiter'
[with T = std::string, C = foo]
    arbiter<string, foo> oops;
    ^~~~~~~~~~~~~~~~~~~~
best.cc:19:14: note: because 'Compares<foo, std::string>' evaluated to false
    requires Compares<C, T>
             ^
```

```
best.cc:15:32: note: because 'c(a, b)' would be invalid: no matching
function for call to object of type 'foo'
    requires(C c, T a, T b) { {c(a, b)} -> std::convertible_to<bool>; };
                               ^
```

Java Generics

Generics were deliberately omitted from the original version of Java. Rather than instantiate containers with different generic parameter types, Java programmers followed a convention in which all objects in a container were assumed to be of the standard base class `Object`, from which all other classes are descended. Users of a container could place any type of object inside. When removing an object, an explicit conversion (what Java calls a `cast`) could be used to reassert the original type. No danger was involved, because objects in Java are self-descriptive, and conversions employ run-time checks.

Though dramatically simpler than the use of templates in C++, this programming convention has three significant drawbacks: (1) users of containers must litter their code with conversions, which many people find distracting or aesthetically distasteful; (2) errors in the use of a container manifest themselves as `ClassCastExceptions` at run time, rather than as compile-time error messages; (3) the error checking of the conversions incurs overhead at run time. Given Java's emphasis on clarity of expression, rather than pure performance, problems (1) and (2) were considered the most serious, and became the subject of a Java Community Process proposal for a language extension in Java 5. The solution adopted is based on the GJ (Generic Java) work of Bracha et al. [BOSW98].

EXAMPLE 7.63
Generic `Arbiter` class in Java

Figure C-7.7 contains a Java version of our `arbiter` class. It differs from the C++ code of Figure C-7.6 in several important ways. First, Java requires that the code for each generic class be manifestly (self-obviously) type safe, independent of any particular instantiation. This means that the type of field `comp`—and in particular, the fact that it provides a `better` method—must be statically declared. As a result, the `Chooser` to be used by a given `Arbiter` instance must be specified as a constructor parameter; it cannot be a generic parameter. (We could have used a constructor parameter in C++; in Java it is mandatory.) For both field `comp` and constructor parameter `c`, we are then faced with the question: what should be the generic parameter of `Chooser`?

The most obvious choice (*not* the one adopted in Figure C-7.7) would be `Chooser<T>`. This would allow us to write

```
class CaseSensitive implements Chooser<String> {
    public boolean better(String a, String b) {
        return a.compareTo(b) < 1;
    }
}
```

```
interface Chooser<T> {
    public boolean better(T a, T b);
}

class Arbiter<T> {
    T bestSoFar;
    Chooser<? super T> comp;

    public Arbiter(Chooser<? super T> c) {
        comp = c;
    }
    public void consider(T t) {
        if (bestSoFar == null || comp.better(t, bestSoFar)) bestSoFar = t;
    }
    public T best() {
        return bestSoFar;
    }
}
```

Figure 7.7 Generic arbiter in Java.

```
...
Arbiter<String> csNames = new Arbiter<String>(new CaseSensitive());
csNames.consider(new String("Apple"));
csNames.consider(new String("aardvark"));
System.out.println(csNames.best());            // prints "Apple"
```

EXAMPLE 7.64

Wildcards and bounds on Java generic parameters

Suppose, however, we were to define

```
class CaseInsensitive implements Chooser<Object> {   // note type!
    public boolean better(Object a, Object b) {
        return a.toString().compareToIgnoreCase(b.toString()) < 1;
    }
}
```

Class `Object` defines a `toString` method (usually used for debugging purposes), so this declaration is valid. Moreover since every `String` is an `Object`, we ought to be able to pass any pair of strings to `CaseInsensitive.better` and get a valid response. Unfortunately, `Chooser<Object>` is not acceptable as a match for `Chooser<String>`. If we typed

```
Arbiter<String> ciNames = new Arbiter<String>(new CaseInsensitive());
```

the compiler would complain. The fix (as shown in Figure C-7.7) is to declare both `comp` and `c` to be of type `<? super T>` instead. This informs the Java compiler that an arbitrary type argument ("`?`") is acceptable as the generic parameter of our `Chooser`, so long as that type is an ancestor of `T`.

```
interface Chooser {
    public boolean better(Object a, Object b);
}

class Arbiter {
    Object bestSoFar;
    Chooser comp;

    public Arbiter(Chooser c) {
        comp = c;
    }
    public void consider(Object t) {
        if (bestSoFar == null || comp.better(t, bestSoFar)) bestSoFar = t;
    }
    public Object best() {
        return bestSoFar;
    }
}
```

Figure 7.8 **Arbiter in Java after type erasure.** Conversions will be inserted by the compiler on calls that return an `Object` or that expect an `Object` to support a particular method.

The `super` keyword specifies a *lower bound* on a type parameter. It is the symmetric opposite of the `extends` keyword, which we used in Example 7.39 to specify an *upper bound*. Together, upper and lower bounds allow us to broaden the set of types that can be used to instantiate generics. As a general rule, we use `extends T` whenever a method returns a `T` object (on which we need to be able to invoke `T` methods); we use `super T` whenever we expect to pass a `T` object as a parameter, but don't mind if the receiver is willing to accept something more general. Given the bounded declarations of Figure C-7.7, our use of `CaseInsensitive` will compile and run just fine:

```
Arbiter<String> ciNames = new Arbiter<String>(new CaseInsensitive());
ciNames.consider(new String("Apple"));
ciNames.consider(new String("aardvark"));
System.out.println(ciNames.best());            // prints "aardvark"
```

Type Erasure

EXAMPLE 7.65

Type erasure and implicit conversions

Generics in Java are defined in terms of *type erasure*: the compiler effectively deletes every generic parameter and argument list, replaces every occurrence of a type parameter with `Object`, and inserts conversions back to concrete types wherever objects are returned from generic methods. The erased equivalent of Figure C-7.7 appears in Figure C-7.8. No conversions are required in this portion of the code. On any use of `best`, however, the compiler would insert an implicit conversions. The statement

```
String winner = csNames.best();
```

will, in effect, be implicitly replaced with

```
String winner = (String) csNames.best();
```

Also, in order to match the `Chooser<String>` interface, our definition of `CaseSensitive` (Example C-7.63) will in effect be replaced with

```
class CaseSensitive implements Chooser {
    public boolean better(Object a, Object b) {
        return ((String) a).compareTo((String) b) < 1;
    }
}
```

■

The advantage of type erasure over the nongeneric (`Object`-based) version of the code is that the programmer doesn't have to write the conversions. In addition, the compiler is able to verify in most cases that the erased code will never generate a `ClassCastException` at run time. The exceptions occur primarily when, for the sake of interoperability with preexisting code, the programmer assigns a generic collection into a nongeneric collection:

EXAMPLE 7.66

Unchecked warnings in Java

```
Arbiter<String> csNames = new Arbiter<String>(new CaseSensitive());
Arbiter alias = csNames;                // nongeneric
alias.consider(Integer.valueOf(3));     // unsafe
```

DESIGN & IMPLEMENTATION

7.11 Why erasure?

Erasure in Java has several surprising consequences. For one, we can't invoke `new T()`, where `T` is a type parameter: the compiler wouldn't know what kind of object to create. Similarly, Java's *reflection* mechanism, which allows a program to examine and reason about the concrete type of an object at run time, knows nothing about generics: `csNames.getClass().toString()` returns `"class Arbiter"`, not `"class Arbiter<String>"`. Why would the Java designers introduce a mechanism with such significant limitations? The answer is backward compatibility or, more precisely, *migration* compatibility, which requires complete interoperability of old and new code.

More so than most previous languages, Java encourages the assembly of working programs, often on the fly, from components written independently by many different people in many different organizations. The Java designers felt it was critical not only that old (nongeneric) programs be able to run with new (generic) libraries, but also that new (generic) programs be able to run with old (nongeneric) libraries. In addition, they took the position that the Java virtual machine, which interprets Java bytecode in the typical implementation, could not be modified. While one can take issue with these goals, once they are accepted erasure becomes a natural solution.

The compiler will issue an "unchecked" warning on the third line of this example, because we have invoked method `consider` on a "raw" (nongeneric) `Arbiter` without explicitly converting the arguments. In this case the warning is clearly warranted: `alias` *shouldn't* be passed an `Integer`. Other examples can be quite a bit more subtle. It should be emphasized that the warning simply indicates the lack of *static* checking; any type errors that actually occur will still be caught at run time. ■

EXAMPLE 7.67

Java generics and built-in types

Note, by the way, that the use of erasure, and the insistence that every instance of a given generic be able to share the same code, means that type arguments in Java must all be descended from `Object`. While `Arbiter<Integer>` is a perfectly acceptable type, `Arbiter<int>` is not. ■

C# Generics

Though generics were omitted from C# version 1, the language designers always intended to add them, and the .NET Common Language Infrastructure (CLI) was designed from the outset to provide appropriate support. As a result, C# 2.0 was able to employ an implementation based on *reification* rather than erasure. Reification creates a different concrete type every time a generic is instantiated with different arguments. Reified types are visible to the reflection library (`csNames.GetType().ToString()` returns `"Arbiter`1[System.Double]"`), and it is perfectly acceptable to call `new T()` if `T` is a type parameter with a zero-argument constructor (a constraint to this effect is required). Moreover where the Java compiler must generate implicit type conversions to satisfy the requirements of the virtual machine (which knows nothing of generics) and to ensure type-safe interaction with legacy code (which might pass a parameter or return a result of an inappropriate type), the C# compiler can be sure that such checks will never be needed, and can therefore leave them out. The result is faster code.

EXAMPLE 7.68

Sharing generic implementations in C#

Of course the C# compiler is free to merge the implementations of any generic instantiations whose code would be the same. Such sharing is significantly easier in C# than it is in C++, because implementations typically employ just-in-time compilation, which delays the generation of machine code until immediately prior to execution, when it's clear whether an identical instantiation already exists somewhere else in the program. In particular, `MyType<Foo>` and `MyType<Bar>` will share code whenever `Foo` and `Bar` are both classes, because C# employs a reference model for variables of class type. ■

EXAMPLE 7.69

C# generics and built-in types

Like C++, C# allows generic arguments to be value types (built-ins or `struct`s), not just classes. We are free to create an object of class `MyType<int>`; we do not have to "wrap" it as `MyType<Integer>`, the way we would in Java. `MyType<int>` and `MyType<double>` would generally not share code, but both would run significantly faster than `MyType<Integer>` or `MyType<Double>`, because they wouldn't incur the dynamic memory allocation required to create a wrapper object, the garbage collection required to reclaim it, or the indirection overhead required to access the data inside. ■

Like Java, C# allows only types as generic parameters, and insists that generics be manifestly type safe, independent of any particular instantiation. It generates

```
interface Chooser<in T> {
    bool better(T a, T b);
}

class Arbiter<T> {
    T bestSoFar;
    Chooser<T> comp;
    bool initialized;

    public Arbiter(Chooser<T> c) {
        comp = c;
        bestSoFar = default(T);
        initialized = false;
    }
    public void Consider(T t) {
        if (!initialized || comp.better(t, bestSoFar)) bestSoFar = t;
        initialized = true;
    }
    public T Best() {
        return bestSoFar;
    }
}
```

Figure 7.9 Generic arbiter in C#.

reasonable error messages if we try to instantiate a generic with an argument that doesn't meet the constraints of the corresponding generic parameter, or if we try, inside the generic, to invoke a method that the constraints don't guarantee will be available.

EXAMPLE 7.70

Generic `Arbiter` class in C#

A C# version of our `Arbiter` class appears in Figure C-7.9. One small difference with respect to Figure C-7.7 can be seen in the `Arbiter` constructor, which must explicitly initialize field `bestSoFar` to `default(T)`. We can leave this out in Java because variables of class type are implicitly initialized to `null`, and type parameters in Java are all classes. In C# `T` might be a built-in or a `struct`, both of which require explicit initialization.

EXAMPLE 7.71

Contravariance in the `Arbiter` interface

A more interesting difference from Figure C-7.7 appears in the definitions of the `Chooser` interface, the `comp` member of class `Arbiter`, and the `c` parameter of the `Arbiter` constructor. In Java, we used explicit lower bounds (`? super T`) on `comp` and `c` to indicate that any `Chooser<S>`, where `S` is a superclass of `T`, would be acceptable. While C# allows us to specify upper bounds in the form of type constraints (we did so in the `sort` routine of Example 7.40), it has no direct equivalent of lower bounds. It does, however, support the related notions of *covariance* and *contravariance*. We have exploited this support in Figure C-7.9, where it appears not as bounds on the `Chooser` passed to a newly created `Arbiter`, but as an `in` modifier on the generic parameter of the `Chooser` interface itself.

The declaration `interface Chooser<in T>` indicates that objects of class `T` will be used only as input parameters to methods of the interface. Suppose now that S is a superclass of T. Since T provides all the methods of S, any method that expects an input of class S will also accept an input of class T. This means that in any context in which all we do is provide T objects as inputs to a Chooser, we can use a "less choosy" Chooser that merely expects S inputs. In other words, Chooser<T> is a superclass of Chooser<S>. Represented graphically,

```
T → S  ⇒  Chooser<S> → Chooser<T>
```

where the → symbol, pronounced "is a," indicates that the item on the left inherits from the item on the right. Chooser<T> is said to be "*contra*variant in T" because the relationship between S and T is reversed when wrapping them in a Chooser.

EXAMPLE 7.72
Covariance

In other situations, objects of a generic type may only be *produced* by the methods of an interface. Consider, for example, the notion of an iterator, as provided by C#'s IEnumerator<T> interface. Method Current of this interface returns an object of class T; no method takes a T object as input. In the C# standard library, the interface is declared as

```
public interface IEnumerator<out T> ...
```

Now suppose again that S is a superclass of T. In any context in which all we do is extract S objects from an IEnumerator, we can use a more specific IEnumerator that gives us T objects instead. In other words, IEnumerator<S> is a superclass of IEnumerator<T>. Graphically,

```
T → S  ⇒  IEnumerator<T> → IEnumerator<S>
```

Here IEnumerator<T> is said to be "*co*variant in T" because the relationship between S and T is preserved when wrapping them in an IEnumerator. In many interfaces, of course, generic parameters appear as both inputs and outputs of methods. For such an interface Foo, there is no subclassing relationship: Foo<T> is said to be "*in*variant in T."

EXAMPLE 7.73
Chooser as a delegate

Returning to the Arbiter example, there is actually a simpler way to write our code in C#. Because the Chooser interface has only a single method, we can express it as a *delegate* instead:

```
delegate bool Chooser<T>(T a, T b);
```

Then in method Arbiter.Consider, we can call the delegate directly as comp(t, bestSoFar). Our new Chooser is roughly analogous to the C declaration

```
typedef _Bool (*Chooser)(T a, T b);
```

(pointer to function of two T arguments, returning a Boolean), except that a C# Chooser object is a closure, not a pointer: it can refer to a static function, a method of a particular object (in which case it has access to the object's fields), or an anonymous nested function (in which case it has access, with unlimited extent, to variables in the surrounding scope). In our particular case, defining Chooser to be a delegate allows us to pass any appropriate function to the Arbiter constructor, without regard to the class inheritance hierarchy. We can declare

```
static bool CaseSensitive(String a, String b) {
    return String.CompareOrdinal(a, b) < 1;
    // use Unicode order, in which upper-case letters come first
}
static bool CaseInsensitive(Object a, Object b) {
    return String.Compare(a.ToString(), b.ToString(), false) < 1;
}
```

and then say

```
Arbiter<String> csNames =
    new Arbiter<String>(new Chooser<String>(CaseSensitive));
csNames.Consider("Apple");
csNames.Consider("aardvark");
Console.WriteLine(csNames.Best());              // prints "Apple"

Arbiter<String> ciNames =
    new Arbiter<String>(new Chooser<String>(CaseInsensitive));
ciNames.Consider("Apple");
ciNames.Consider("aardvark");
Console.WriteLine(ciNames.Best());              // prints "aardvark"
```

The compiler is perfectly happy to instantiate `CaseInsensitive` as a `Chooser <String>`, because `String`s can be passed as `Object`s. ■

CHECK YOUR UNDERSTANDING

48. Why was it difficult, historically, to produce high-quality error messages for misuses of C++ templates? How do the *concepts* of C++20 address this problem?

49. What is the purpose of explicit instantiation in C++? What is the purpose of `extern` templates?

50. What is *template metaprogramming*?

51. Explain the difference between *upper bounds* and *lower bounds* in Java type constraints. Which of these does C# support?

52. What is *type erasure*? Why is it used in Java?

53. Under what circumstances will a Java compiler issue an "unchecked" generic warning?

54. Why must fields of generic parameter type be explicitly initialized in C#?

55. For what two main reasons are C# generics often more efficient than comparable code in Java?

56. Summarize the notions of *covariance* and *contravariance* in generic types.

57. How does a C# *delegate* differ from an interface with a single method (e.g., the C++ `chooser` of Figure C-7.6)? How does it differ from a function pointer in C?

7 Type Systems

7.7 Exercises

7.27 C++ has no direct analogue of the `extends X` and `super X` clauses of Java. Why not?

7.28 Write a simple abstract `ordered_set<T>` class (an *interface*) whose methods include `void insert(T val)`, `void remove (T val)`, `bool lookup (T val)`, and `bool is_empty()`, together with a language-appropriate iterator, as described in Section 6.5.3. Using this abstract class as a base, build a simple `list_set` class that uses a sorted linked list internally. Try this exercise in C++, Java, and C#. Note that you will need constraints on `T` in Java and C#. You may also want them in C++. Discuss the differences among your implementations.

7.29 Building on the previous exercise, implement higher-level `union<T>`, `intersection<T>`, and `difference<T>` functions that operate on ordered sets. Note that these should not be members of the `ordered_set<T>` class, but rather stand-alone functions: they should be independent of the details of `list_set` or any other particular `ordered_set`. So, for example, `union(A, B, C)` should verify that `A` is empty, and then add to it all the elements found in `B` or `C`. Explain, for each of C++, Java, and C#, how to handle the comparison of elements.

7.30 Continuing Example C-7.63, the call

```
csNames.consider(null);
```

will generate a run-time exception, because `String.compareTo` is not designed to take `null` arguments.

(a) Modify Figure C-7.7 to guard against this possibility by including a predicate `public Boolean valid(T a);` in the `Chooser<T>` interface, and by modifying `consider` to make an appropriate call to this predicate. Modify class `CaseSensitive` accordingly.

(b) Suggest how to make similar modifications to the C# `Arbiter` of Figure C-7.9 and Example C-7.70. How should you handle lower bounds when you need both `Better` and `Valid`?

7.31 (a) Modify your solution to Exercise 7.15 so that the comparison routine is an explicit generic parameter, reminiscent of the `chooser` of Figure C-7.6.

(b) Give an alternative solution in which the comparison routine is an extra parameter to `sort`.

7.32 Consider the C++ program shown in Figure C-7.10. Explain why the final call to `first_n` generates a compile-time error, but the call to `last_n` does not. (Note that `first_n` is generic but `last_n` is not.) Show how to modify the final call to `first_n` so that the compiler will accept it.

7.33 Consider the following code in C++:

```
template <typename T>
class cloneable_list : public list<T> {
public:
    cloneable_list<T>* clone() {
        auto rtn = new cloneable_list<T>();
        for (auto e : *this) {
            rtn->push_back(e);
        }
        return rtn;
    }
};

...
cloneable_list<foo> L;
...
cloneable_list<foo>* Lp = L.clone();
```

Here `*Lp` will be a "deep copy" of `L`, containing a copy of each `foo` object. Try to write equivalent code in Java. What goes wrong? How might you get around the problem?

```
#include <iostream>
#include <list>
using std::cout;
using std::list;

template<typename T> void first_n(list<T> p, int n) {
    for (typename list<T>::iterator li = p.begin(); li != p.end(); li++) {
        if (n-- <= 0) break;
        cout << *li << " ";
    }
    cout << "\n";
}

void last_n(list<int> p, int n) {
    for (list<int>::reverse_iterator li = p.rbegin(); li != p.rend(); li++) {
        if (n-- <= 0) break;
        cout << *li << " ";
    }
    cout << "\n";
}

class int_list_box {
    list<int> content;
public:
    int_list_box(list<int> l) { content = l; }
    operator list<int>() { return content; }
        // user-supplied operator for coercion/conversion
};

int main() {
    int i = 5;
    list<int> l;

    for (int i = 0; i < 10; i++) l.push_back(i);
    int_list_box b(l);

    first_n(l, i);      // works
    last_n(b, i);       // works (coerces b)
    first_n(b, i);      // static semantic error
}
```

Figure 7.10 **Coercion and generics in C++.** The compiler refuses to accept the final call to `first_n`.

7 Type Systems

7.8 Explorations

7.44 Learn more about *concepts* in C++, together with the earlier notions of *named requirements* and the “substitution failure is not an error” (SFINAE) idiom. Compare and contrast concepts with the constraint mechanisms of Java and C#.

7.45 Explore the support for generics in Scala, Eiffel, Ada, or some other programming language. Compare this support to that of C++, Java, and C#. What might account for the differences? Which approach(es) do you prefer? Why?

7.46 Explore more fully the concepts of *covariance* and *contravariance* in object-oriented languages, as exemplified by the `in` and `out` modifiers for generic parameters in C# 4.0. Discuss the connection between these concepts and the notions of upper and lower bounds on generic parameters (`? extends T` and `? super T` in Java).

8 Composite Types

8.1.4 Unions (Variant Records, Datatypes)

EXAMPLE 8.82

Nested structs and unions in traditional C

A variant record provides two or more alternative fields or collections of fields, only one of which is valid at any given time. This notion has its roots in the `equivalence` statement of Fortran I and in the `union` types of Algol 68. Building on the `element` type of Example 8.1, one could specify a variant record as follows in (pre-2011) C:

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
    _Bool naturally_occurring;
    union {
        struct {
            char *source;
                /* textual description of principal commercial source */
            double prevalence;
                /* fraction, by weight, of Earth's crust */
        } natural_info;
        double lifetime;
            /* half-life in seconds of the most stable known isotope */
    } extra_fields;
} copper;
```

Here the programmer presumably intends for the `naturally_occurring` field to indicate which parts of the union are currently valid. A `true` value indicates that the element has at least one naturally occurring stable isotope; in this case fields `source` and `prevalence` are intended to describe how the element may be obtained and how commonly it occurs. A `false` value indicates that the element results only from atomic collisions or the decay of heavier elements; in this case, field `lifetime` is intended to indicate how long atoms so created tend to survive before undergoing radioactive decay. These mutually exclusive sets of fields (`source` and

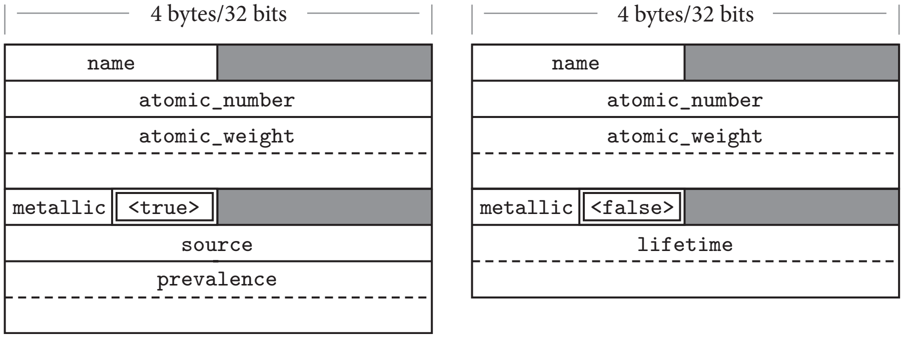


Figure 8.16 Likely memory layouts for `element` variants. The value of the `naturally_occurring` field (shown here with a double border) is intended to indicate which of the interpretations of the remaining space is valid. Field `source` is assumed to point to a string that has been independently allocated.

`prevalence`, on the one hand, or `lifetime` on the other) are sometimes known as *variants*. Either the first or the second variant may be useful, but never both at once. From an implementation perspective, nonoverlapping uses suggest that the variants may share space, as shown in Figure C-8.16.

One significant problem with our nested `struct` and `union` is the need for two extra levels of naming. While the always-present fields can be accessed as, say, `copper.atomic_weight`, fields of the inner `struct` are much less easy to name: `copper.extra_fields.natural_info.source`.

EXAMPLE 8.83

A variant record in Pascal

Pascal's principal contribution to union types was to integrate them with records. In Pascal syntax, our running example might look like this:

```
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean;
    case naturally_occurring : Boolean of
      true : (
        source : string_ptr;
        prevalence : real;
      );
      false : (
        lifetime : real;
      )
end;
```

Here the `naturally_occurring` field is introduced with the keyword `case`, to formalize its role as a *tag* or *discriminant*. Note that the variant fields have no extraneous levels of naming: we can refer directly to `copper.source`.

EXAMPLE 8.84

Anonymous unions in C11 and C++11

Leveraging an extension long supported by `gcc`, C11 and its successors allow a nameless (*anonymous*) struct or union to appear within another struct or union. The members of the anonymous construct are then directly visible in the surrounding context:

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
    _Bool naturally_occurring;
    union {
        struct {
            char *source;
            double prevalence;
        };
        double lifetime;
    };
} copper;
...
copper.source = "various ores";
```

Anonymous nesting makes variants in C11 as convenient as those of Pascal. C++11 even allows anonymous unions in non-struct contexts:

```
void foo() {
    union {
        int a;
        int b;
    };
    ...
    a = 3;
```

Safety

EXAMPLE 8.85

Breaking type safety with unions

A potentially more significant problem with unions in C is the lack of type safety. Mistakes in which the programmer writes to one field of a union and then reads from the other are relatively common:

```
union {
    int i;
    double d;
} u;
...
u.d = 3.0;
...
printf("%d", u.i);
```

Here the `printf` statement, which attempts to output `i` as an integer, will (in most implementations) take its bits from the floating-point representation of 3.0—almost certainly a mistake, but one that the language implementation will not catch. ■

To avoid these sorts of errors, Algol 68 included features to track the status of unions at run time, and to prevent access to currently invalid fields. Similar features can be found in Ada and in ML-family languages today. Our running `element` example might be written as follows in OCaml:

EXAMPLE 8.86

Type-safe unions in OCaml

```
type natural_info = {source : string; prevalence : float};;
type synthesized_info = {lifetime : float};;
type extra_info =
  | Natural of natural_info
  | Synthesized of synthesized_info;;

type element = {
    name : string;
    atomic_number : int;
    atomic_weight : float;
    metallic : bool;
    extra_fields : extra_info};;
```

As in traditional C, the variant portions of a record introduce extra levels of nesting in OCaml. To enforce correct usage, the language implementation maintains a hidden tag in every union object, to indicate which variant is currently valid. Values can be declared only as aggregates that specify the tag and all the fields:

```
let copper = {
    name = "Cu";
    atomic_number = 29;
    atomic_weight = 63.546;
    metallic = true;
    extra_fields = Natural ({
        source = "various ores and native deposits";
        prevalence = 0.00005
    })
};;
```

Individual fields can be read, but only in the context of a `match` expression that verifies the value of the tag:

```
exception Union_error;;
let source (e : element) =
    match e.extra_fields with
      | Natural     n -> n.source
      | Synthesized _ -> raise Union_error;;

let copper_source = source copper;;
```

■

Variant records with mandatory tags (explicit or hidden) are known as *discriminated unions*. Variant records without tags (as in C) are known as *nondiscriminated unions*. Pascal provided both, but in the absence of an analogue of `match`, even the discriminated case was difficult to implement safely (more on this in Exercise C-8.38). Ada, by contrast, combines syntax reminiscent of Pascal with the type safety of ML.

Variants in Ada

EXAMPLE 8.87
Ada variants and tags (discriminants)

Ada variant records must always have a tag (called the *discriminant*). Language rules ensure that this tag can never be changed without simultaneously assigning values to all of the fields of the corresponding variant. The assignment can occur either via whole-record assignment (e.g., `a := b`, where `a` and `b` are variant records), or via assignment of an aggregate (e.g., `p := (polar => true, rho => 1.0, theta => pi/2.0))`. In addition to appearing as a field within the record, the discriminant of a variant record in Ada must also appear in the header of the record's declaration:

```
type Element (Naturally_Occurring : Boolean := True) is record
    Name : String (1..2);
    Atomic_Number : Integer;
    Atomic_Weight : Long_Float;
    Metallic : Boolean;
    case Naturally_Occurring is
        when True =>
            Source : String_Ptr;
            Prevalence : Long_Float;
        when False =>
            Lifetime : Long_Float;
    end case;
end record;
```

Here we have not only declared the discriminant of the record in its header, we have also specified a default value for it. A declaration of a variable of type `Element` has the option of accepting this default value:

```
Copper : Element;
```

or overriding it:

```
Plutonium : Element (False);
Neptunium : Element (Naturally_Occurring => False);
    -- alternative syntax
```

If the type declaration for `Element` did not specify a default value for `Naturally_Occurring`, then all variables of type `Element` would have to provide a value. These rules guarantee that the tag field of a variant record is never uninitialized.

An Ada record variable whose declaration specifies a value for the discriminant is said to be *constrained*. Its tag field can never be changed by a subsequent assignment. This immutability means that the compiler can allocate just enough space to hold the specified variant; this space may in some cases be significantly smaller than would be required for other variants. A variable whose declaration does not provide an initial value for the discriminant is said to be *unconstrained*. Its tag will be initialized to the value in the type declaration, but may be changed by later (whole-record) assignments, so the space that the record occupies must be large enough to hold any possible variant.

EXAMPLE 8.88

A discriminated subtype in Ada

An Ada subtype definition can also constrain the discriminant(s) of its parent type:

```
subtype Natural_Element is Element (True);
```

Variables of type `Natural_Element` will all be constrained; their `Naturally_Occurring` field cannot be changed. Because `Natural_Element` is a subtype, rather than a derived type, values of type `Element` and `Natural_Element` are compatible with each other, though a run-time semantic check will usually be required to assign the former into the latter. ■

EXAMPLE 8.89

Discriminated array in Ada

Ada uses record discriminants not only for variant tags, but in general for any value that affects the size of a record. Here is an example that uses a discriminant to specify the length of an array:

DESIGN & IMPLEMENTATION

8.14 The placement of variant fields

To facilitate space saving in constrained variant records, Ada requires that all variant parts of a record appear at the end. This rule ensures that every field has a constant offset from the beginning of the record, with no holes (in any variant) other than those required for alignment. When a constrained variant record is elaborated, the Ada run-time system need only allocate sufficient space to hold the specified variant, which is never allowed to change. Pascal had a similar rule, designed for a similar purpose. When a variant record was allocated from the heap in Pascal (via the built-in `new` operator), the programmer had the option of specifying `case` labels for the variant portions of the record. A record so allocated was never allowed to change to a different variant, so the implementation could allocate precisely the right amount of space.

Modula-2, which did not provide `new` as a built-in operation, eliminated the ordering restriction on variants. All variables of a variant record type had to be large enough to hold any variant. The usual implementation assigned a fixed offset to every field, with holes following small internal variants as necessary. Similar conventions apply to `unions` and `structs` in modern C.

```
type Element_Array is array (Integer range <>) of Element;
type Alloy (Num_Components : Integer) is record
    Name : String (1..30);
    Components : Element_Array (1..Num_Components);
    Tensile_Strength : Long_Float;
end record;
```

The `<>` notation in the initial definition of `Element_Array` indicates that the bounds are not statically known. Further discussion of dynamic arrays appears in Section 8.2.2. As with discriminants used for variant tags, the programmer must either specify a default value for the discriminant in the type declaration (we did not do so above), or else every declaration of a variable of the type must specify a value for the discriminant (in which case the variable is constrained, and the discriminant cannot be changed).

The Object-Oriented Alternative

EXAMPLE 8.90

Derived types as an alternative to unions

In dropping variant records from their parent language, the designers of Modula-3 noted [Har92, p. 110] that much of the same effect could be obtained with classes and inheritance. Oberon, similarly, replaced variants with a more general mechanism for *type extension* (sidebar 10.3), and the designers of Java and C# dropped the `union`s of C and C++. In place of the C code of Example C-8.82, a Java programmer might write

```
class Element {
    public String name;
    public int atomicNumber;
    public double atomicWeight;
    public boolean metallic;
}
class NaturalElement extends Element {
    public String source;
    public double prevalence;
}
class SyntheticElement extends Element {
    public double lifetime;
}
```

Like the discriminated subtypes of Ada, this approach constrains each object to a single variant at creation time, but this may not be a problem: while the class of a particular object never changes, class-type variables are references in Java and C#. A variable of type `Element` can easily refer to an object of class `NaturalElement` or `SyntheticElement` at run time.

✓ **CHECK YOUR UNDERSTANDING**

55. What are *anonymous* unions and structs? What purpose do they serve? How is this related to the integration of variants with records in Pascal and its descendants?

56. What is a *tag* (*discriminant*) in a variant record? In a language like Ada or OCaml, how does it differ from an ordinary field?

57. Discuss the type safety problems that arise with variant records. How can these problems be addressed?

58. Summarize the rules that prevent access to inappropriate fields of variant records in OCaml and Ada.

59. Why might one wish to *constrain* a variable, so that it can hold only one variant of a type?

60. Explain how classes and inheritance can be used to obtain the effect of constrained variant records.

8 Composite Types

8.5.3 Dangling References

Memory access errors—dangling references, memory leaks, out-of-bounds access to arrays—are among the most common program bugs, and among the most difficult to find. Testing and debugging techniques for memory errors vary in when they are performed, how much they cost, and how conservative they are. Several commercial and open-source tools employ binary instrumentation (Section 16.2.3) to track the allocation status of every block in memory and to check every load or store to make sure it refers to an allocated block. These tools have proved to be highly effective, but they can slow a program several-fold, and may generate *false positives*—indications of error in programs that, while arguably poorly written, are technically correct. Many compilers can also be instructed to generate dynamic semantic checks for certain kinds of memory errors. Such checks must generally be fast (much less than 2× slowdown), and must never generate false positives. In this section we consider two candidate implementations of checks for dangling references.

Tombstones

EXAMPLE 8.91

Dangling reference detection with tombstones

Tombstones [Lom75, Lom85] allow a language implementation to catch all dangling references, to objects in both the stack and the heap. The idea is simple: rather than have a pointer refer to an object directly, we introduce an extra level of indirection (Figure C-8.17). When an object is allocated in the heap (or when a pointer is created to an object in the stack), the language run-time system allocates a tombstone. The pointer contains the address of the tombstone; the tombstone contains the address of the object. When the object is reclaimed, the tombstone is modified to contain a value (typically zero) that cannot be a valid address. To avoid special cases in the generated code, tombstones are also created for pointers to static objects.

For heap objects, it is easy to invalidate a tombstone when the program calls the deallocation operation. For stack objects, the language implementation must be able to find all tombstones associated with objects in the current stack frame

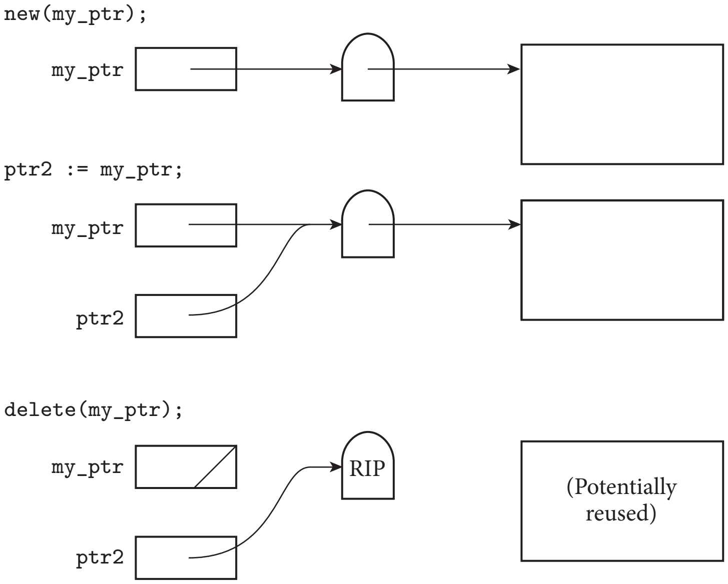


Figure 8.17 **Tombstones.** A valid pointer refers to a tombstone that in turn refers to an object. A dangling reference refers to an "expired" tombstone.

when returning from a subroutine. One possible solution is to link all stack-object tombstones together in a list, sorted by the address of the stack frame in which the object lies. When a pointer is created to a local object, the tombstone can simply be added to the beginning of the list. When a pointer is created to a parameter, the run-time system must scan down the list and insert in the middle, to keep it sorted. When a subroutine returns, the epilogue portion of the calling sequence invalidates the tombstones at the head of the list, and removes them from the list.

Tombstones may be allocated from the heap itself or, more commonly, from a separate pool. The latter option avoids fragmentation problems, and makes allocation relatively fast, since the first tombstone on the free list is always the right size.

Tombstones can be expensive, both in time and in space. The time overhead includes (1) creation of tombstones when allocating heap objects or using a "pointer to" operator, (2) checking for validity on every access, and (3) double-indirection. Fortunately, checking for validity can be made essentially free on most machines by arranging for the address in an "invalid" tombstone to lie outside the program's address space. Any attempt to use such an address will result in a hardware interrupt, which the operating system can reflect up into the language run-time system. We can also use our invalid address, in the pointer itself, to represent the constant `nil`. If the compiler arranges to set every pointer to `nil` at elaboration time, then the hardware will catch any use of an uninitialized pointer. (This technique works without tombstones, as well.)

The space overhead for tombstones can be significant. The simplest approach is never to reclaim them. Since a tombstone is usually significantly smaller than the object to which it refers, a program will waste less space by leaving a tombstone around forever than it would waste by never reclaiming the associated object. Even so, any long-running program that continually creates and reclaims objects will eventually run out of space for tombstones. A potential solution, which we consider in Section 8.5.4, is to augment every tombstone with a *reference count*, and reclaim tombstones themselves when the reference count goes to zero.

Tombstones have a valuable side effect. Because of double-indirection, it is easy to change the location of an object in the heap. The run-time system need not locate every pointer that refers to the object; all that is required is to change the address in the tombstone. The principal reason to change heap locations is for *storage compaction*, in which all dynamically allocated blocks are "scooted together" at one end of the heap in order to eliminate external fragmentation. Tombstones are not widely used in language implementations, but the Macintosh operating system (versions 9 and below) used them internally, for references to system objects such as file and window descriptors. They also closely resemble the implementation used for *smart pointers* in the C++ standard library.

Locks and Keys

Locks and *keys* [FL80] are an alternative to tombstones. Their disadvantage is that they provide only probabilistic protection from dangling pointers. Their advantage is that they avoid the need to keep tombstones around forever (or to figure out when to reclaim them). Again the idea is simple: Every pointer is a tuple consisting of an address and a key. Every object in the heap begins with a lock. A pointer to an object in the heap is valid only if the key in the pointer matches the lock in the object (Figure C-8.18). When the run-time system allocates a new heap object, it generates a new key value. These can be as simple as serial numbers, but should avoid "common" values such as zero and one. When an object is reclaimed, its lock is changed to some arbitrary value (e.g., zero) so that the keys in any remaining pointers will not match. If the block is subsequently reused for another purpose, we expect it to be very unlikely that the location that used to contain the lock will be restored to its former value by coincidence. The original implementation of locks and keys in Pascal considered only pointers to heap objects, as outlined above, but in principle a run-time system could also add locks to objects allocated on the stack.

EXAMPLE 8.92
Dangling reference detection with locks and keys

Like tombstones, locks and keys incur significant overhead. They add an extra word of storage to every pointer and to every block in the heap. They increase the cost of copying one pointer into another. Most significantly, they incur the cost of comparing locks and keys on every access (or every access that cannot be proven to be redundant). It is unclear whether the lock and key check is cheaper or more expensive than the tombstone check. A tombstone check may result in two cache misses (one for the tombstone and one for the object); a lock and key check is unlikely to cause more than one. On the other hand, the lock and key check requires a significantly longer instruction sequence on most machines. To

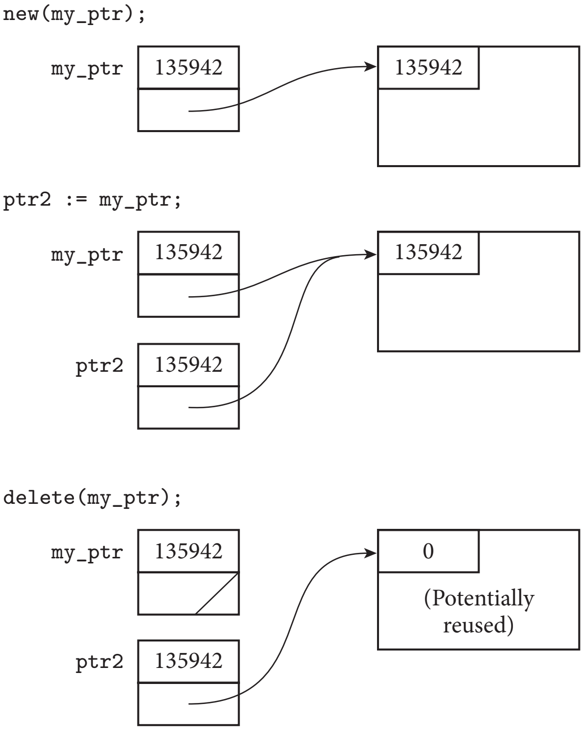


Figure 8.18 **Locks and keys.** A valid pointer contains a key that matches the lock on an object in the heap. A dangling reference is unlikely to match.

minimize time and space overhead, most compilers do not by default generate code to check for dangling references.

CHECK YOUR UNDERSTANDING

61. What are *tombstones*? What changes do they require in the code to allocate and deallocate memory, and to assign and dereference pointers?
62. Explain how tombstones can be used to support *compaction*.
63. What are *locks* and *keys*? What changes do they require in the code to allocate and deallocate memory, and to assign and dereference pointers?
64. Explain why the protection afforded by locks and keys is only probabilistic.
65. Discuss the comparative advantages of tombstones and locks and keys as a means of catching dangling references.

8 Composite Types

8.7 Files and Input/Output

The first two subsections below are devoted to interactive and file-based I/O, respectively. Section C-8.7.3 then considers the common special case of text files.

8.7.1 Interactive I/O

On a modern machine, interactive I/O usually occurs through a graphical user interface (GUI: "gooey") system, with a mouse, a keyboard, and a bit-mapped screen that in turn support windows, menus, scrollbars, buttons, sliders, and so on. GUI characteristics vary significantly among, say, Microsoft Windows, the Macintosh, and Unix's X11; the differences are one of the principal reasons it is difficult to port applications across platforms.

Within a single platform, the facilities of a GUI system usually take the form of library routines (to create or resize a window, print text, draw a polygon, and so on). Input events (mouse move, button push, keystroke) may be placed in a queue that is accessible to the program, or tied to *event handler* subroutines that are called by the run-time system when the event occurs. Because the handler is triggered from outside, its activities must generally be *synchronized* with those of the main program, to make sure that both parties see a consistent view of any data shared between them. We will discuss events further in Section 9.6, and synchronization in Section 13.3.

A few programming languages—notably Smalltalk—attempt to incorporate a standard set of GUI mechanisms into the language itself. The Smalltalk design team was part of the original group at Xerox's Palo Alto Research Center (PARC) that invented mouse-and-window based interfaces in the early 1970s. Unfortunately, while the Smalltalk GUI is successful within the confines of the language, it tends not to integrate well with the "look and feel" of the host system on which it runs.

Other languages—Java, for example—provide graphics as a standard library package. Java's original GUI facilities (the Abstract Window Toolkit—AWT) had something of a "least common denominator" look to them. The Java routines and their interface have evolved significantly over time; the more recent Swing and JavaFX libraries have "pluggable" look and feel, allowing them to integrate more easily with (and port more easily among) a variety of window systems.

The "parallel execution" of the program and the human user that characterizes interactive systems is difficult to capture in a functional programming model. A functional program that operates in a "batch" mode (taking its input from a file and writing its output to a file) can be modeled as a function from input to output. A program that interacts with the user, however, requires a very concrete notion of program ordering, because later input may depend on earlier output. If both input and output take the form of an ordered sequence of tokens, then interactive I/O can be modeled using lazy data structures, a subject we considered in Section 6.6.2. More general solutions can be based on the notion of *monads*, which use a functional notion of sequencing to model side effects. We will consider these issues again in Sections 11.5 and 11.8.

8.7.2 File-Based I/O

Persistent files are the principal mechanism by which programs that run at different times communicate with each other. A few language proposals (e.g., Argus [LS83] and χ [SH92]) allow ordinary variables to persist from one invocation of a program to the next, and a few experimental operating systems (e.g., Opal [CLFL94] and Hemlock [GSB$^+$93]) provide persistence for variables outside the language proper. In addition, some language-specific programming environments, such as those for Smalltalk and Common Lisp, provide a notion of *workspace* that includes persistent named variables. With coming advances in nonvolatile memory technology, such features may find their way into a larger number of languages. Historically, they have been more the exception than the rule. For the most part, data that need to outlive a particular program invocation have needed to reside in files.

Like interactive I/O, files can be incorporated directly into the language, or provided via library routines. In the latter case, it is still a good idea for the language designers to suggest a standard library interface, to promote portability of programs across platforms. The lack of such a standard in Algol 60 is widely credited with impeding the language's widespread use. One of the principal reasons to incorporate I/O into the language proper is to make use of special syntax. In particular, several languages, notably Fortran and Pascal, provide built-in I/O facilities in order to obtain type-safe "subroutines" that take a variable number of parameters, some of which may be optional.

Depending on the needs of the programmer and the capabilities of the host operating system, data in files may be represented in binary form, much as it is in memory, or as *text*. In a binary file, the number 1066_{10} would be represented by the 32-bit value 10000101010_2. In a text file, it would probably be represented by the character string `"1066"`. Temporary files are usually kept in binary form for the sake of speed and convenience. Persistent files are commonly kept in both forms. Text files are more easily ported across systems: issues of word size, byte order, alignment, floating-point format, and so on do not arise. Text files also have the advantage of human readability: they can be manipulated by text editors and related tools. Unfortunately, text files tend to be large, particularly when used to hold numeric data. A double-precision floating-point number occupies only eight bytes in binary form, but can require as many as 24 characters in decimal notation (not counting any surrounding white space). Text files also incur the cost of binary to text conversion on output, and text to binary conversion on input. The size problem can be addressed, at least for archival storage, by using data compression. Mechanisms to control text/binary conversion tend to be the most complicated part of I/O; we discuss them in the following subsection.

EXAMPLE 8.93
Files as a built-in type

When I/O is built into a language, files are usually declared using a built-in type constructor, as for example in Pascal:

```
var my_file : file of foo;
```

EXAMPLE 8.94
The `open` operation

If I/O is provided by library routines, the library usually provides an opaque type to represent a file. In either case, each file variable is generally bound to an external, operating system–supported file by means of an *open* operation. In C, for example, one says

```
my_file = fopen(path_name, mode);
```

The first argument to `fopen` is a character string that names the file, using the naming conventions of the host operating system. The second argument is a string that indicates whether the file should be readable, writable, or both, whether it should be created if it does not yet exist, and whether it should be overwritten or appended to if it does exist.

EXAMPLE 8.95
The `close` operation

When a program is done with a file, it can break the association between the file variable and the external object by using a *close* operation:

```
fclose(my_file);
```

In response to a call to `close`, the operating system may perform certain "finalizing" operations, such as unlocking an exclusive file (so that it may be used by other programs), rewinding a tape drive, or forcing the contents of buffers out to disk.

Most files, both binary and text, are stored as a linear sequence of characters, words, or records. Every open file then has a notion of *current position*: an implicit reference to some element of the sequence. Each `read` or `write` operation implicitly advances this reference by one position, so that successive operations access successive elements, automatically. In a *sequential* file, this automatic advance is the only way to change the current position. Sequential files usually correspond to media like printers and tapes, in which the current position has a physical representation (how many pages we've printed; how much tape is on each spool) that is difficult to change.

In other, *random-access* files, the programmer can change the current position to an arbitrary value by issuing a *seek* operation. In a few programming languages (e.g., Cobol and PL/I), random-access files (also called *direct* files) have no notion of current position. Rather, they are *indexed* on some key, and every `read` or `write` operation must specify a key. A file that can be accessed both sequentially *and* by key is said to be *indexed sequential*.

Random-access files usually correspond to media like solid-state flash drives or magnetic or optical disks, in which the current position can be changed with relative ease. Where tape drives (still widely used for archival storage) can take more than a minute to seek to a given position, modern disks take anywhere from 5 to 200 ms, depending on technology. (Note that 5 ms is still a very long time—10 million cycles on a 2 GHz processor.) Seeking on a solid-state device is essentially instantaneous. A few languages—notably Pascal—provide no random-access files, though individual implementations may support random access as a nonstandard language extension.

8.7.3 Text I/O

It is conventional to think of text files as consisting of a sequence of *lines*, each of which in turn consists of characters. In older systems, particularly those designed around the metaphor of punch cards, lines are reflected in the organization of the file itself. A `seek` operation, for example, may take a line number as argument. More commonly, a text file is simply a sequence of characters. Within this sequence, control (nonprinting) characters indicate the boundaries between lines. Unfortunately, end-of-line conventions are not standardized. In Unix and in modern versions of the Mac OS, each line of a text file ends with a *newline* ("control-J") character, ASCII value 10. (On "classic" Macs, each line ended with a *carriage return* ("control-M") character, ASCII value 13.) On Windows machines, each line ends with a carriage return/newline pair. Text files are usually sequential.

Despite the muddied conventions for line breaks, text files are much more portable and readable than binary files.[1] Because they do not mirror the structure of internal data, text files require extensive conversions on input and output. Issues to be considered include the base for integer values (and the representation of nondecimal bases); the representation of floating-point values (number of digits, placement of decimal point, notation for exponent); the representation of enumerations and other nonnumeric, nonstring types; and positioning, if any, within columns (right and left justification, zero or white-space fill, "floating" dollar signs in Cobol). Some of these issues (e.g., the number of digits in a floating-point

[1] We are speaking here, of course, of plain-text ASCII or Unicode files. So-called "rich text" files, consisting of formatted text in particular fonts, sizes, and colors, perhaps with embedded graphics, are another matter entirely. Word processors typically represent rich text with a combination of binary and ASCII data, though ASCII-only standards such as Postscript, textual PDF, RTF, and XML can be used to enhance portability.

number) are influenced by the hardware, but most are dictated by the needs of the application and the preferences of the programmer.

In most languages the programmer can take complete control of input and output formatting by writing it all explicitly, using language or library mechanisms to read and write individual characters only. I/O at such a low level is tedious, however, and most languages also provide more high-level operations. These operations vary significantly in syntax and in the degree to which they allow the programmer to specify I/O formats. We illustrate the breadth of possibilities with examples from four imperative languages: Fortran, Ada, C, and C++.

Text I/O in Fortran

EXAMPLE 8.96

Formatted output in Fortran

In Fortran, we could write a character string, an integer, and an array of 10 floating-point numbers as follows:

```
character s*20
integer n
real r (10)
...
write (4, '(A20, I10, 10F8.2)'), s, n, r
```

In the `write` statement, the 4 indicates a *unit number*, which identifies a particular output file. The quoted, parenthesized expression is called a *format*; it specifies how the printed variables are to be represented. In this case, we have requested a 20-column ASCII string, a 10-column integer, and 10 eight-column floating-point numbers (with two columns of each reserved for the fractional part of the value). Fortran provides an extremely rich set of these *edit descriptors* for use inside of formats. Cobol, PL/I, and Perl provide comparable facilities, though with a very different syntax.

EXAMPLE 8.97

Labeled formats

Fortran allows a format to be specified indirectly, so it may be used in more than one input or output statement:

```
write (4, 100), s, n, r            ! 100 is the line number
...                                ! of the format statement
100 format (A20, I10, 10F8.2)
```

It also allows formats to be created at run time, and stored in strings:

```
character(len=20) :: fmt
...
fmt = "(A20, I10, 10F8.2)"
...
write (4, fmt), s, n, r
```

If the programmer does not know, or does not care about, the precise allocation of columns to fields, the format can be omitted:

```
write (4, *), s, n, r
```

In this case, the run-time system will use default format conventions.

EXAMPLE 8.98
Printing to standard output

To write to the standard output stream (i.e., the terminal or its surrogate), the programmer can use the `print` statement, which resembles a `write` without a unit number:

```
print*, s, n, r          ! * means default format
```

For input, `read` is used both for standard input and for specific files; in the former case, the unit number is omitted, together with the extra set of parentheses:

```
read 100, s, n, r
...
read*, s, n, r           ! * means default format
```

The star may be omitted in Fortran 90.

In the full form of `read`, `write`, and `print`, additional arguments may be provided in the parenthesized list with the unit number and format. These can be used to specify a variety of additional information, including a label to which to jump on end-of-file, a label to which to jump on other errors, a variable into which to place status codes returned by the operating system, a set of labels (a "namelist") to attach to the output values, and a control code to override the usual automatic advance to the next line of the file. Because there are so many of these optional arguments, most of which are usually omitted, they are usually specified using *named* (keyword) parameter notation, a notion we defer to Section 9.3.3.

The variety of shorthand versions of `read`, `write`, and `print`, together with the fact that they operate on a variable number of program variables, makes it very difficult to cast them as "ordinary" subroutines. Fortran 90 provides optional and named parameters, but Fortran 77 does not, and even in Fortran 90 there is no way to define a subroutine with an *arbitrary* number of parameters.

Text I/O in Ada

Ada provides a suite of five standard library packages for I/O. The `Sequential_IO` and `Direct_IO` packages are for binary files. They provide generic file types that can be instantiated for any desired element type. The `IO_Exceptions` and `Low_Level_IO` packages handle error conditions and device control, respectively. The `Text_IO` package provides formatted input and output on sequential files of characters.

EXAMPLE 8.99
Formatted output in Ada

Using `Text_IO`, our original three-variable Fortran output statement would look something like this in Ada:

```
s : array (1..20) of Character;
n : Integer;
r : array (1..10) of Real;
...
```

```
set_output(My_File);
Put(N, 10);
Put(S);
for I in 1..10 loop Put(R(I), 5, 2); end loop;
New_Line;
```

In the Put of an element of R (within the loop), the second parameter specifies the number of digits before the decimal point, rather than the width of the entire number (including the decimal point), as it did in Fortran. The Put of S will use the string's natural length. If a different length is desired, the programmer will have to write blanks or Put a substring explicitly. If precise output positioning is not desired for the integers and real numbers, the extra parameters in their Put calls can be omitted; in this case the run-time system will use standard defaults. The programmer can use additional library routines to change these defaults if desired. A call to Set_Output invokes a similar mechanism: it changes the default notion of output file. ■

EXAMPLE 8.100
Overloaded Put routines

There are two overloaded forms of Put for every built-in type. One takes a file name as its first argument; the other does not. The last five lines above could have been written

```
Put(My_File, N, 10);
Put(My_File, S);
for I in 1..10 loop Put(My_File, R(I), 5, 2); end loop;
New_Line(My_File);
```

The programmer can of course define additional forms of Get and Put for arbitrary user-defined types. All of these facilities rely on standard Ada mechanisms; in contrast to Fortran, no support for I/O is built into the language itself. ■

Text I/O in C

C provides I/O through a library package called stdio; as in Ada, no support for I/O is built into the language itself. Many C implementations, however, build knowledge of I/O functions into the compiler, so it can issue warnings when arguments appear to be used incorrectly.

EXAMPLE 8.101
Formatted output in C

Our example output statement would look something like this in C:

```
char s[20];
int n;
double r[10];
...
fprintf(my_file, "%20s%10d", s, n);
for (i = 0; i < 10; i++) fprintf(my_file, "%8.2f", r[i]);
fprintf(my_file, "\n");
```

The arguments to fprintf are a file, a format string, and a sequence of expressions. The format string has capabilities similar to the formats of Fortran, though the syntax is very different. In general, a format string consists of a sequence of characters with embedded "placeholders," each of which begins with a percent sign. The placeholder `%20s` indicates a 20-character string; `%d` indicates an integer in decimal notation; `%8.2f` indicates an 8-character floating-point number, with two digits to the right of the decimal point.

EXAMPLE 8.102

Text in format strings

As in Fortran, formats can be computed and stored in strings, and a single `fprintf` statement can print an arbitrary number of expressions. As in Ada, an explicit `for` loop is needed to print an array. Commonly the format string also contains labeling text and white space:

```
strcpy(s, "four");                    /* copy "four" into s */
n = 20;
char *fmt = "%s score and %d years ago\n";
fprintf(my_file, fmt, s, n);
```

A percent sign can be printed by doubling it:

```
fprintf(my_file, "%d%%\n", 25);     /* prints "25%" */
```

EXAMPLE 8.103

Formatted input in C

Input in C takes a similar form. The `fscanf` routine takes as argument a file, a format string, and a sequence of pointers to variables. In the common case, every argument after the format is a variable name preceded by a "pointer to" operator:

```
fscanf(my_file, "%s %d %lf", s, &n, &r[0]);
```

In this call, the `%s` placeholder will match a string of maximal length that does not include white space. If this string is longer than 20 characters (the length of `s`), then `fscanf` will write beyond the end of the storage for the string. (This weakness in `scanf` is one of the sources of the so-called "buffer overflow" bugs discussed in Sidebar 8.7. It can be avoided in this example by replacing the `%s` specifier with `%19s`, which will cause `fscanf` to move at most 19 bytes, plus a terminating `NUL`.) The three-character `%lf` placeholder informs the library routine that the corresponding argument is a `double`; the 2-character sequence `%f` would read into a `float`.[2] Accidentally using a placeholder for the wrong size variable is a common error in older implementations of C; forgetting the ampersand on a trailing argument is another. While such mistakes will often be caught by a modern C compiler with special-case knowledge of `fscanf`, they would always be caught in a language with type-safe I/O. Note that we have read a single element of `r`; as with `fprintf`, a `for` loop would be needed to read the whole array. Note also that while

[2] C's `doubles` are double-precision IEEE floating-point numbers in most implementations; `floats` are usually single precision. The lack of safety for `%s` arguments is only one of several problems with `fscanf`. Others include the inability to "skip over" erroneous input, and undefined behavior when there is insufficient input. Instead of `fscanf`, seasoned C programmers tend to use `fgets`, which reads (length-limited) input into a string, followed by manual parsing using `strtol` (string-to-`long`), `strtod` (string-to-`double`), and so on.

we have not made use of this fact in our example, `fscanf` returns an integer value indicating the number of `&`-identified items that were scanned successfully. ■

We have noted above that the I/O routines of Fortran and Pascal are built into the language largely to permit them to take a variable number of arguments. We have also noted that moving I/O into a library in Ada forces us to make a separate call to `put` for every output expression. So how do `fprintf` and `fscanf` work? It turns out that C permits functions with a variable number of parameters (we will discuss such functions in more detail in Section 9.3.3). Unfortunately, the types of trailing parameters are unspecified, which makes compile-time type checking of variable-length argument lists impossible in the general case. Moreover, the lack of run-time type descriptors in C precludes run-time checking as well. At the same time, because the C library (including `fprintf` and `fscanf`) is part of the language standard, special knowledge of these routines can be built into the compiler—and often is: while the I/O routines of C are formally defined as "ordinary" functions, they are typically implemented in the same way as their analogues in Fortran and Pascal. As a result, C compilers will often provide good error diagnostics when the arguments to `fprintf` or `fscanf` do not match the format string.

To simplify I/O to and from the standard input and output streams, `stdio` provides routines called `printf` and `scanf` that omit the initial arguments of `fprintf` and `fscanf`. To facilitate the formatting of strings *within* a program, `stdio` also provides routines called `sprintf` and `sscanf`, which replace the initial arguments of `fprintf` and `fscanf` with a pointer to an array of characters. The `sscanf` function "reads" from this array; `sprintf` "writes" to it. Fortran 90 provides similar support for intraprogram formatting through so-called *internal files*.

Text I/O in C++

As a descendant of C, C++ supports the `stdio` library described in the previous subsection. It also supports an I/O library called `iostream` that exploits the object-oriented features of the language. The `iostream` library is more flexible than `stdio`, provides arguably more elegant syntax (though this is a matter of taste), and is completely type safe.

C++ *streams* use operator overloading to co-opt the << and >> symbols normally used for bit-wise shifts. The `iostream` library provides an overloaded version of << and >> for each built-in type, and programmers can define versions for new types. To print a C-style character string in C++, one writes

EXAMPLE 8.104
Formatted output in C++

```
my_stream << s;
```

To output a string and an integer one can write

```
my_stream << s << n;
```

This idiom requires that `my_stream` be an instance of the `ostream` (output stream) class defined in the `iostream` library. The << operator, with a right operand of type

`T`, is then syntactic sugar for either the "operator function" `operator<<(ostream, T)` or the "operator method" `ostream::operator<<(T)`, as described in Section 3.5.2. As it turns out, `iostream` provides an operator function for C-style strings and a member function for integers. Because << associates left-to-right, the statement above is equivalent to

```
(operator<<(my_stream, s)).operator<<(n);
```

The code works because `operator<<` returns a reference to its first argument as its result (as we shall see in Section 9.3.1, C++ supports both a value model and a reference model for variables).

EXAMPLE 8.105
Stream manipulators

As shown so far, output to an `ostream` uses default formatting conventions. To change conventions, one may embed so-called *stream manipulators* in a sequence of << operations. To print `n` in octal notation (rather than the default decimal), we could write

```
my_stream << oct << n;
```

To control the number of columns occupied by `s` and `n`, we could write

```
my_stream << setw(20) << s << setw(10) << n;
```

The `oct` manipulator causes the stream to print all subsequent numeric output in octal. The `setw` manipulator causes it to print its next string or numeric output in a field of a specified minimum width (behavior reverts to the default after a single output).

The `oct` manipulator is declared as a function that takes an `ostream` as a parameter and produces a reference to an `ostream` as its result. Because it is not followed by empty parentheses, the occurrence of `oct` in the output sequence above is *not* a call to `oct`; rather, a reference to `oct` is passed to an overloaded version of << that expects a manipulator function as its right-hand argument. This version of << then calls the function, passing the stream (the left-hand argument of <<) as argument.

The `setw` manipulator is even trickier. It is declared as a function that returns a reference to what we might call an "object closure"—an object containing a reference to a function and a set of arguments. In this particular case, `setw(20)` is a call to a *constructor* function that returns a closure containing the number 20 and a pointer to the `setw` manipulator. (We will discuss constructors in detail in Section 10.3, and object closures in Section 3.6.3.) The `iostream` library provides an overloaded version of << that expects an object closure as its right-hand argument. This version of << calls the function inside the closure, passing it as arguments the stream (the left-hand argument of <<) and the integer inside the closure.

EXAMPLE 8.106
Array output in C++

The `iostream` library provides a wealth of manipulators to change the formatting behavior of an `ostream`. Because C++ inherits C's handling of pointers and arrays, however, there is no way for an `ostream` to know the length of an array. As a result, our full output example still requires a `for` loop to print the `r` array:

```
char s[20];
int n;
double r[10];
...
my_stream << setw(20) << s << setw(10) << n;
for (i = 0; i < 10; i++)
    my_stream << setiosflags(ios::fixed)
        << setw(8) << setprecision(2) << r[i];
my_stream << "\n";
```

Here the manipulators in the output sequence in the `for` loop specify fixed format (rather than scientific) for floating-point numbers, with a field width of eight, and two digits past the decimal point. The `setiosflags` and `setprecision` manipulators change the default format of the stream; the changes apply to all subsequent output. ■

EXAMPLE 8.107

Changing default format

To avoid calling stream manipulators repeatedly, we could modify our example as follows:

```
my_stream.flags(my_stream.flags() | ios::fixed);
my_stream.precision(2);
for (i = 0; i < 10; i++) my_stream << setw(8) << r[i];
```

To facilitate the restoration of defaults, the `flags` and `precision` functions return the previous value:

```
ios::fmtflags old_flags = my_stream.flags(my_stream.flags() | ios::fixed);
int old_precision = my_stream.precision(2);
for (i = 0; i < 10; i++) my_stream << setw(8) << r[i];
my_stream.flags(old_flags);
my_stream.precision(old_precision);
```

■

Formatted input in C++ is analogous to formatted output. It uses `istreams` instead of `ostreams`, and the >> operator instead of <<. It also supports a suite of manipulators comparable to those for output. I/O on the standard input and output streams does not require different functions; the programmer simply begins an input or output sequence with the standard stream name `cin` or `cout`. (In keeping with C tradition, there is also a standard stream `cerr` for error messages.) To support intraprogram formatting of character strings, the `strstream` library provides `istrstream` and `ostrstream` object classes that are derived from `istream` and `ostream`, and that allow a stream variable to be bound to a string instead of to a file.

✓ **CHECK YOUR UNDERSTANDING**

66. Explain the differences between interactive and file-based I/O, between temporary and persistent files, and between binary and text files. (Some of this information is in the main text.)

67. What are the comparative advantages of *text* and *binary* files?

68. Describe the end-of-line conventions of Unix, Windows, and Macintosh files.

69. What are the advantages and disadvantages of building I/O into a programming language, as opposed to providing it through library routines?

70. Summarize the different approaches to text I/O adopted by Fortran, Ada, C, and C++.

71. Describe some of the weaknesses of C's `scanf` mechanism.

72. What are *stream manipulators*? How are they used in C++?

8 Composite Types

8.9 Exercises

8.35 In Example 6.70 we described a programming idiom in which an iterator takes a "loop body" function as argument, and applies it to every element of a given container or set. Show how to use this idiom in ML to apply a function to every element of the tree in Example 11.39. Write versions of your iterator for preorder, inorder, and postorder traversals.

8.36 Show how unions can be used in C to interpret the bits of a value of one type as if they represented a value of some other type. Explain why the same technique does not work in Ada. After consulting an Ada manual, describe how an `unchecked` pragma can be used to get around the Ada rules.

8.37 Are variant records a form of polymorphism? Why or why not?

8.38 Learn the details of variant records in Pascal.

(a) You may have noticed that the language does not allow you to pass the tag field of a variant record to a subroutine by reference. Why not?

(b) Explain how you might implement dynamic semantic checks to catch references to uninitialized fields of a tagged variant record. Changing the value of the tag field should cause all fields of the variant part of the record to become uninitialized. Suppose you want to avoid adding flag fields within the record itself (e.g., to avoid changing the offsets of fields in a systems program). How much harder is your task?

(c) Explain how you might implement dynamic semantic checks to catch references to uninitialized fields of an *untagged* variant record. Any assignment to a field of a variant should cause all fields of other variants to become uninitialized. Any assignment that changes the record from one variant to another should also cause all other fields of the new variant to be uninitialized. Again, suppose you want to avoid adding flag fields within the untagged record itself. How much harder is your task?

8.39 We noted in Section C-8.1.4 that Ada requires the variant portions of a record to occur at the end, to save space when a particular record is constrained to have a comparatively small variant part. Could a compiler rearrange fields to achieve the same effect, without the restriction on the declaration order of fields? Why or why not?

8.40 Reference counts can be used to reclaim tombstones,. While it is certainly possible to create a circular structure with tombstones, the fact that the programmer is responsible for explicit deallocation of heap objects implies that reference counts will fail to reclaim tombstones only when the programmer has failed to reclaim the objects to which they refer. Explain how to leverage this observation to catch memory leaks at run time. Does your solution work in all cases? Explain.

8.41 For objects allocated in the heap, we have suggested that a "lock" for dangling reference detection be allocated in the object header, at a known offset from the beginning of the object itself. This choice doesn't work well for pointers to static or stack-allocated objects, or in general for pointers created with an "address of" (`&`) operator, since these may refer to fields in the middle of larger objects.

Zhou [ZCH23] has suggested solving this problem by adding an extra field to every pointer, to indicate the offset between the lock and the pointed-to object. A single lock can then protect an entire stack frame, or all the static objects in a module.

Elaborate on this suggestion. Specifically, describe the code that must be executed in subroutine prologues and epilogues—and on pointer assignment and dereference—in order to detect dangling references in a language that permits pointers to non-heap objects.

8.42 In Section 8.5.4 we introduced the notion of *smart pointers* in C++. Learn how these are implemented, and write an explanation. Discuss the relationship to tombstones.

8.43 Rewrite Example C-8.103 using `fgets`, `strtol`, `strtod`, etc. (read the `man` pages), so that it is guaranteed not to result in buffer overflow.

8.44 The output routines of several languages (e.g., `println` in Swift) give special treatment to ends of lines. By contrast, C's `printf` does not; it treats newlines and carriage returns the same as any other character. What are the comparative advantages of these approaches? Which do you prefer? Why?

8 Composite Types

8.10 Explorations

8.54 Research the history of smart pointers (Section 8.5.4) in C++, including the `unique_ptr`, `shared_ptr`, and `weak_ptr` of C++11; the `auto_ptr` of C++98, and the various pointer classes of the popular Boost library. How has the standard set of pointers evolved over time? What accounts for the changes? Do you consider the current mechanisms an adequate replacement for automatic garbage collection? Why or why not?

8.55 Find a Cobol manual and learn about the language's facilities for text I/O. Prepare a written comparison of those facilities to those of the languages described in Section C-8.7.3.

8.56 If you were designing the text I/O facilities for a new programming language, what approach would you take? In particular, do you believe that I/O should be a built-in part of the language, or should it be handled by library routines?

9 Subroutines and Control Abstraction

9.2.1 Displays

EXAMPLE 9.67

Nonlocal access using a display

As noted in the main text, a display is an embedding of the static chain into an array. The *j*th element of the display contains a reference to the frame of the most recently active subroutine at lexical nesting level *j*. The first element of the display is thus a reference to the frame of some subroutine *S* nested directly inside the main program; the second element is a reference to the frame of a routine that is nested inside of *S*, and so forth, until we reach the currently active routine. Figure C-9.7 contains an example. ■

If the display is stored in memory, then a nonlocal object can be loaded into a register with two memory accesses: one to load the display element into a register, the second to load the object. On a machine with a large number of registers, one might be tempted to reduce the overhead to only one memory access by keeping the entire display in registers, but that would probably be a bad idea: display elements tend to be accessed much less frequently than other things (e.g., local variables) that might be kept in the registers instead.

Maintaining the Display

Maintenance of a display is slightly more complicated than maintenance of a static chain, but not by much. Perhaps the most obvious approach would be to maintain the static chain as usual, and simply fill the display at procedure entry and exit, by walking down the chain. In most cases, however, the following (much faster) scheme suffices: when calling a subroutine at lexical nesting level *j*, the callee saves the current value of the *j*th display element into the stack, and then replaces that element with a copy of its own (newly created) frame pointer. Before returning, it restores the old element. Why does this mechanism work? As with static chains, there are two cases to consider:

1. The callee is nested (directly) inside the caller. In this case the caller and the callee share all display elements up to the current level. Putting the callee's frame pointer into the display simply extends the current level by one. It is conceivable that the old value needn't be saved, but in general there is no way

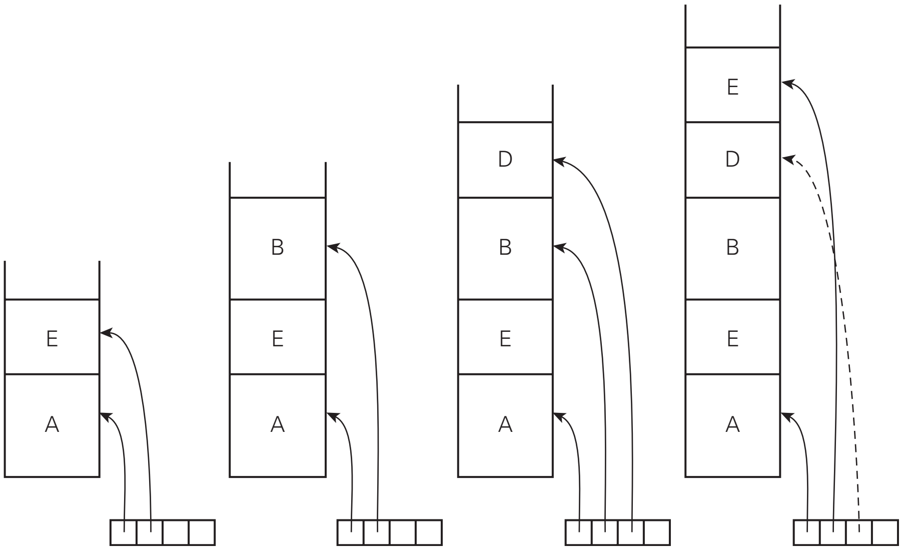


Figure 9.7 **Nonlocal access using a display.** The stack configurations, from left to right, illustrate the contents of the display (at bottom) for a sequence of subroutine calls, assuming the lexical nesting of Figure 9.1. Display elements beyond that of the currently executing subroutine are not used.

to tell. The caller itself might have been called by code that is very deeply nested, and that is counting on the integrity of a very deep display, in which case the old display element *will* be needed. A smart compiler may be able to avoid the save in certain circumstances.

2. The callee is at lexical nesting level j, $k \geq 0$ levels out from the caller. In this case the caller and callee share all display elements up through $j - 1$. The caller's

DESIGN & IMPLEMENTATION

9.9 Lexical nesting and displays

Because the display is a fixed-size array, compilers that use a display to implement access to nonlocal objects generally impose a limit (the size of the display) on the maximum depth to which subroutines may be nested. If this limit is larger than, say, five or six, it is unlikely that any programmer will ever wish for more. Note that the display does not eliminate the need for a frame pointer. Because local variables are accessed so often, it is important to have the address of the current frame in a register, where it can be used for displacement-mode addressing. Similarly, on a RISC processor, where a 32-bit address will not fit in one instruction, it is important to maintain a base register for the most commonly accessed global variables as well.

entry at level j is different from the callee's, so the callee must save it before storing its own frame pointer. If the callee in turn calls a routine at level $j + 1$, that routine will change another element of the display, but all old elements will be restored before they are needed again.

If the callee is a leaf routine then the display can be left intact; no one will use the element corresponding to the callee's nesting level before control returns to the caller.

Closures

A subroutine that is passed as a parameter, stored in a variable, or returned from a function must be called through some sort of *closure* (Section 3.6) that captures the referencing environment. In a language implementation based on static chains, a closure can be represented as a ⟨code address, static link⟩ pair. Displays are not as simple. A standard technique is to create two "entry points"—starting addresses—for every subroutine. One of these is for "normal" calls, the other for calls through closures. When a closure is created, it contains the address of the alternative entry point. The code at that entry point saves elements 1 through j of the display into the stack (it will have to create a larger-than-normal stack frame in order to do this), and then replaces those elements with values taken from (or calculated from) the closure. The alternative entry then makes a nested call to the main body of the subroutine (it skips the code immediately following the normal entry—the code that creates the normal stack frame and updates the display). When the subroutine returns, it comes back to the code of the alternative entry, which restores the old value of the display before returning to the actual caller.

More space-conserving implementations of display-based closures are possible (see Exercise C-9.29), but with higher run-time overhead.

Comparison to Static Chains

In general, maintaining a display is slightly more expensive than maintaining a static chain, though the comparison is not absolute. In the usual case, passing a static link to a called routine requires $k \geq 0$ load instructions in the caller, followed by one store instruction in the callee (to place the static link at the appropriate offset in the stack frame). The store may be skipped in leaf routines, assuming that a register is available to hold the link as long as it is needed. No overhead is required to maintain the static chain when returning from a subroutine. With a display, a nonleaf callee requires two loads and three stores (1 + 2 in the prologue and 1 + 1 epilogue) to save and restore display elements. Because the callee does all the work, displays may save a little bit on code size, compared to static chains. As noted above, displays significantly complicate the creation and use of closures.

The original advantage of displays—reduced cost for access to objects in outer scopes—seems less clear today than once it did. In fact, while displays were popular in the CISC compilers of the 1970s and 1980s, they are less common in recent compilers. Most programs don't nest subroutines more than two or three levels deep, so static chains are seldom very long, and variables in surrounding scopes tend not to be accessed very often. If they *are* accessed often, common subexpression optimizations (to be discussed in Chapter 17) are likely to ensure that a pointer to the appropriate frame remains in a register.

Some language designers have argued that the development of object-oriented programming (the subject of Chapter 10) has eliminated the need for nested subroutines [Han81]. Others might even say that the success of C has shown such routines to be unneeded. Without nested subroutines, of course, the choice between static chains and displays is moot.

✓ CHECK YOUR UNDERSTANDING

46. Describe how we access an object at lexical nesting level *k* in a language implementation based on displays.

47. Why isn't the display typically kept in registers?

48. Explain how to maintain the display during subroutine calls.

49. What special concerns arise when creating closures in a language implementation that uses displays?

50. Summarize the tradeoffs between displays and static chains. Describe a program for which displays will result in faster code. Describe another for which static chains will be faster.

9 Subroutines and Control Abstraction

9.2.2 Stack Case Studies: LLVM on Arm; `gcc` on x86

To make stack management a bit more concrete, we present a pair of case studies: Apple's LLVM-based C compiler for the iPhone (Arm) and the GNU compiler suite for 32- and 64-bit x86. Both examples follow the general scheme outlined in Section 3.2.2, with differences in details that reflect the history of the respective compilers and the architecture of the target machines.

LLVM on Arm

An overview of the Arm instruction set architecture (ISA) can be found in Section C-5.4.5. For the sake of interoperability, Arm Ltd. publishes a standard for subroutine calling sequences that allows code from different vendors and compilers to link and run together. The standard has several variants, reflecting hardware features (Thumb mode, floating-point or vector instructions and registers, dynamic linking) that may or may not be present on a given processor or in its software environment. We focus here on the conventions adopted by Apple's C compiler for the iPhone and iPad (version 4.2), at optimization level `-O2`. The Apple compiler uses the 32-bit Arm back end (version 3.2svn) of the LLVM compiler suite. Given the level of detail in Arm's standard, code produced by other compilers is likely to be quite similar. Note, however, that the conventions for 64-bit code are very different; they are not documented here.

As noted in Section C-5.4.5, register `r14` (also known as `lr`) is special-cased by the ISA to receive the return address in subroutine call (`bl`—branch-and-link) instructions. Register `r13` (also known as `sp`) is similarly reserved for use as the stack pointer. It is not modified by `bl` instructions, but several variants of `push` and `pop`, which do update `sp`, are commonly part of the subroutine calling sequence. Some compilers for Arm, though not all, dedicate a third register by convention for use as a frame pointer; LLVM uses `r7` for this purpose.

EXAMPLE 9.68
LLVM/Arm stack layout

A typical LLVM/Arm stack frame appears in Figure C-9.8. The `sp` register points to the *last used* location in the stack (note that some compilers aim the pointer at

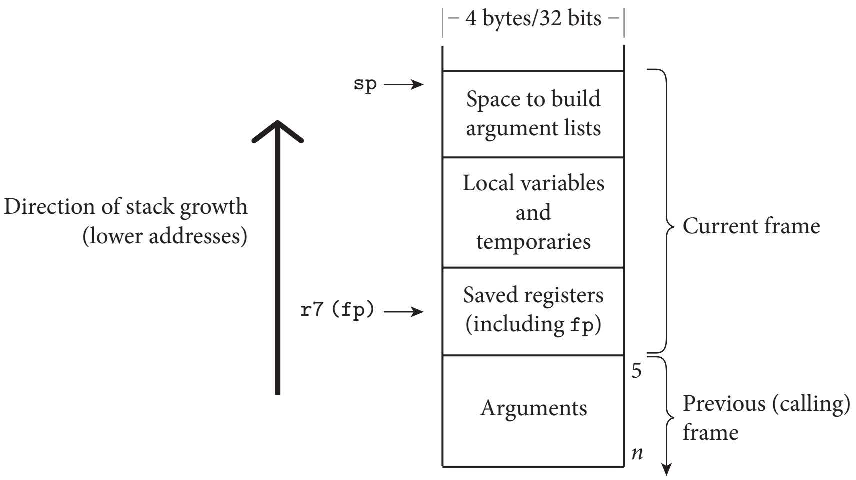

4 bytes/32 bits
sp
Space to build argument lists
Local variables and temporaries
Saved registers (including fp)
Arguments
r7 (fp)
Direction of stack growth (lower addresses)
Current frame
Previous (calling) frame
5
n

Figure 9.8 **Layout of the subroutine call stack for Apple's LLVM-based C compiler for Arm, running in 32-bit mode.** As in Figure 9.2, lower addresses are toward the top of the page.

the *first unused* location). Arm's subroutine calling standard requires that the stack always be word-aligned (sp $\bmod 4 = 0$). At an external call (to a subroutine in a different compilation unit) it must be double-word aligned (sp $\bmod 8 = 0$).

The first four arguments to a subroutine are always passed in registers. Additional arguments may be passed on the stack, with the last argument in the deepest location. Space for stack-based arguments is considered a part of the calling routine. If the current routine is not a leaf, space for any stack-based arguments it needs to pass to additional routines is preallocated, at the top (lowest-addressed-end) of the frame, as part of the subroutine prologue.

Space for local variables and for any temporary values that will not fit in registers is allocated in the middle of the frame. If the subroutine ever applies an address-of operator (& in C) to a low-numbered argument (one that will have been passed in a register), or if it passes such an argument to another routine by reference, the compiler creates a local variable to hold the argument, and initializes it with the value passed in the register.

Any callee-saves registers that may be overwritten by the current routine are saved at the bottom of the frame, directly beyond any stack-based arguments. The frame pointer (r7) is typically among these. LLVM arranges for the current fp to point to the location of the saved fp. ■

Argument Passing Conventions Arguments and locals of the current subroutine are accessed via offsets from the fp. Arguments in the process of being passed to the next routine are assembled at the top of the frame, and are accessed via offsets from the sp. The first four arguments are passed in registers r0–r3. A double-precision floating-point number is divided into two 32-bit halves, and passed as if it were two integers. (Some other Arm compilers pass floating-point arguments in the floating-point registers.) Records (structs) that appear early in the argument list may also be split into 32-bit pieces, and passed in multiple registers. An argument may be split between registers and the stack, if part but not all of it will fit in registers.

The argument build area at the top of the frame is designed to be large enough to hold the largest argument list that may be passed to any called routine. This convention may waste a bit of space in certain cases, but it ensures that arguments never need to be "pushed" in the usual sense of the word: the sp does not change when they are placed into the stack.

Return values up to 4 bytes in length occupy register r0. Double-word scalar return values occupy register pair r0–r1; quad-word scalar return values occupy registers r0–r3. Record-type return values of more than four bytes are placed in memory, at a location chosen by the caller and passed as an extra, hidden argument. If the return value is to be assigned immediately into a variable (e.g., x = foo()), the caller can simply pass the address of the variable. If the value is to be passed in turn to another subroutine, the caller can pass the appropriate address within its own argument build area. (Writing the return value into this space will probably destroy the returning function's own arguments, but that's fine in the absence of call-by-value/result: at this point the arguments are no longer needed.) Finally, though one doesn't see this idiom often (and most languages don't support it), C allows the caller to extract a field directly from the return value of a function (e.g., x = foo().a + y;); in this case the caller must pass the address of a temporary location within the "local variables and temporaries" part of its stack frame.

DESIGN & IMPLEMENTATION

9.10 Leveraging pc = r15

Because Arm assigns a register number to the program counter, that counter can be read and written (almost) like any other register. Writes to the pc cause a branch in control. This convention, together with the choice of lr = r14 and pc = r15, enables an interesting optimization. If a subroutine is not a leaf (i.e., it calls another routine), lr will be among the registers saved at the bottom of the frame. If we suppose, for concreteness, that the subroutine plans to overwrite callee-saves registers r4 and r5, and we know that we need to update the frame pointer (r7), then the subroutine prologue is likely to contain a push {r4, r5, r7, lr} instruction. This instruction stores the registers in sorted order, with the highest-numbered register (in this case, lr) at the highest address—deepest in the stack. One might naturally expect the epilogue to contain a symmetric pop {r4, r5, r7, lr} instruction, followed immediately by bx lr (branch to location in lr). But since the pc and lr have adjacent register numbers, the compiler can—and typically does—achieve the same result with a single pop {r4, r5, r7, pc} instruction.

Arm and Thumb Mode Switching One of the more unusual features of the 32-bit Arm ISA (as described in Section C-5.4.5) is the presence of two separate instruction encodings. As on most RISC machines, the A32 encoding represents each instruction with 32 bits. The alternative T32 encoding, also known as "Thumb," represents the most common instructions in only 16 bits; the resulting improvement in code density can be important in embedded applications. While the two encodings are quite different (and in particular, T32 is not a subset of A32), program fragments that use different encodings can be linked into a single program.

To switch from one format to another, the program uses special `bx` (branch and exchange instruction set) and `blx` (branch with link and exchange instruction set) instructions. When the target address is statically known, the assumption is that the programmer knows that the source and target encodings are different, so the processor needs to change modes in the course of performing the branch. When the target address is in a register (as it will be when returning from a subroutine, or when calling through a pointer, a virtual method table, or a closure), Arm exploits the fact that instructions never appear at an odd address (T32 instructions are always word aligned; A32 instructions are always longword aligned). Because the least significant bit of the target address must always be 0, this bit can be used in the register to specify the target instruction set: 0 means A32; 1 means T32.

EXAMPLE 9.69
LLVM/Arm calling sequence

Calling Sequence Details The calling sequence to maintain the LLVM/Arm stack is as follows. The caller

1. saves (into the "local variables and temporaries" part of its frame) any caller-saves registers whose values are still needed
2. puts up to four small arguments (or "chunks" of larger arguments) into registers `r0–r3`
3. stores the remaining arguments into the argument build area at the top of the current frame
4. performs a `bl` or `blx` instruction, which puts the return address in register `lr`, jumps to the target address, and optionally changes instruction set encoding

On 32-bit Arm, the caller-saves registers are just the ones that are used for arguments—namely, `r0–r3`. In a language with nested subroutines (not supported by Apple's compiler), the caller would need to place the static link into another register immediately before performing the `bl` or `blx`.

In its prologue, the callee

1. pushes any necessary registers onto the stack
2. initializes the frame pointer by adding an appropriate small constant to the `sp`, placing the result in `r7`
3. subtracts enough from the `sp` to make space for local variables, temporaries, and the argument build area at the top of the stack, rounding down to a lower address if necessary to ensure that these objects have appropriate alignment

Saved registers include (a) the frame pointer, r7 (assuming the current routine needs a frame pointer of its own); (b) any callee-saves registers (r4–r6 and r8–r11) whose values may be changed before returning; and (c) the link register, lr, if the current routine is not a leaf, or if it uses lr as an additional temporary.

In its epilogue, immediately before returning, the callee

1. places the function return value (if any) into r0–r3 or memory, as appropriate
2. subtracts a small constant from r7, placing the result in sp; this effectively deallocates the bulk of the frame
3. pops saved registers from the stack, with the pc taking the place held by lr in the corresponding save in the prologue; this has the side effect of branching back to the caller (see Sidebar C-9.10)

Finally, if appropriate, the caller moves the return value to wherever it is needed. Caller-saves registers are restored lazily over time, as their values are needed.

To support the use of symbolic debuggers, the compiler generates a wealth of symbol table information, in the open-source DWARF format [DWA17]. It embeds this information into the object file. The information is most accurate when the program is compiled without any code improvement (-O0). For each subroutine, the information includes the starting and ending addresses of the routine; the name, type, and location (register name or frame pointer offset) of every formal parameter and local variable; the set of instructions corresponding to each line of source code; the size and layout of the stack frame; and a list of which registers were saved. ■

gcc on x86

To illustrate the differences among compilers and architectures, our second case study considers the GNU compiler collection (gcc, version 4.8.1) on the x86. We begin with 32-bit code and then explain the differences that obtain on 64-bit machines. Our example again focuses mostly on C, which acts as sort of a "lowest common denominator" among high-level languages. We also consider nested subroutines and closures, however, since these appear in some of the collection's supported languages.

An overview of the x86-32 ISA appears in Section C-5.4.5. Given the machine's CISC heritage and the comparatively small number of registers (only six are available for general-purpose use), all arguments are passed on the stack when running in 32-bit mode. To give the compiler the freedom to evaluate arguments out of order when desired, recent versions of gcc employ an argument build area similar to that of the LLVM case study. Unlike LLVM, recent versions of gcc omit the use of a separate frame pointer by default, making register ebp (rbp in 64-bit mode) available for other purposes; exceptions occur when specified by the programmer (using the -no-omit-frame-pointer command-line switch), when compiling at optimization levels -O0 and -O1, when a subroutine has a local variable whose size is not known at compile time (Figure 8.7), or when a subroutine calls alloca (a legacy mechanism to create temporary space within the current stack frame).

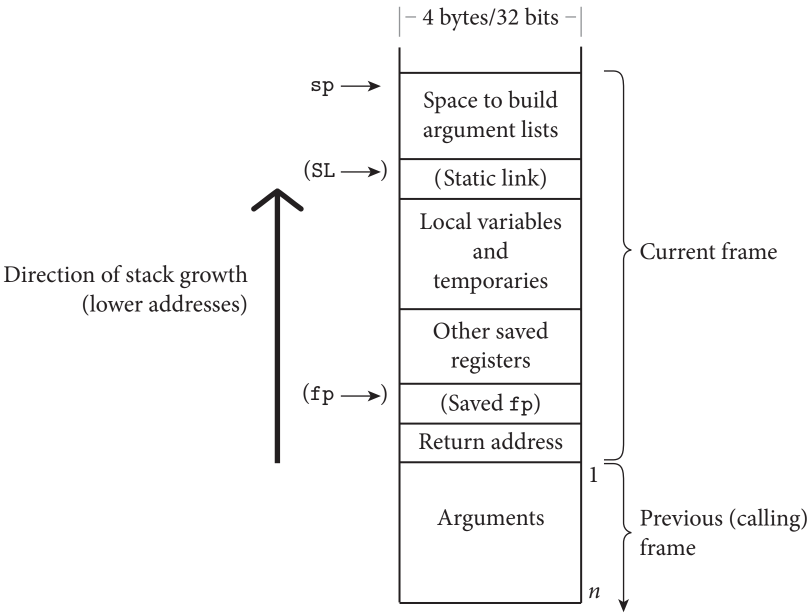


Figure 9.9 Layout of the subroutine call stack for the GNU Compiler Collection (`gcc`) on 32-bit x86. The return address is present in all frames. All other parts of the frame are optional; they are present only if required by the current subroutine. In x86 terminology, the `sp` is named `esp`; the `fp` is `ebp` (extended base pointer). The static link, in languages with nested subroutines, is passed in register `ecx`. `SL` marks the location that will be referenced by the static link (if any) of any subroutine nested immediately inside this one. A routine that is neither innermost nor outermost will save its own static link at the location referenced by the static link of its children.

Historically, omission of the frame pointer made it difficult or even impossible for symbolic debuggers to perform a "backtrace" operation (identifying the frames of calling routines), but this limitation has been removed with modern debugging standards like DWARF.

Calling sequences for the x86 vary from vendor to vendor, and have evolved considerably over time, as changes in microarchitecture changed performance tradeoffs. Most modern sequences use the `call` and `ret` instructions. The former pushes the return address onto the stack, updating the `sp`, and branches to the called routine. The latter pops the return address off the stack, again updating the `sp`, and branches back to the caller. Several additional, more complex instructions, retained for backward compatibility, are typically not generated by modern compilers, because they were designed for calling sequences with an explicit display and without an argument build area, or because they don't pipeline as well as equivalent sequences of simpler instructions.

EXAMPLE 9.70
`gcc`/x86-32 stack layout

Argument Passing Conventions Figure C-9.9 shows a stack frame for the x86-32. As in the LLVM case study, the `sp` points to the last used location on the stack.

Arguments in the process of being passed to another routine are accessed via offsets from the sp; everything else is accessed via offsets from the fp, if present—otherwise the sp. All arguments are passed in the stack. In languages (Ada, in particular) that permit nested subroutines, register ecx is used to pass the static link. If the current routine has at least one lexically nested child and is itself lexically nested in some parent, then a copy of the static link will be saved into the stack just above (at a lower address than) the area used for local variables and temporaries. When a nested routine is running, its own static link will point to the saved link in this current routine, or to the local variables and temporaries, if this current routine is outermost.

Functions return integer or pointer values in register eax. Floating-point values are returned in the first of the "x87" floating-point registers, st(0). Composite values (records, arrays, etc.) of 8 bytes or less are returned in the register pair eax–edx, as are "long long" (64-bit) integers. For larger return values (records, arrays, etc.), the compiler passes a hidden first argument (on the stack) whose value is the address at which the return value should be written. ■

EXAMPLE 9.71

gcc/x86-32 calling sequence

Calling Sequence Details The calling sequence to maintain the gcc/x86-32 stack is as follows. The caller

1. saves (into the "local variables and temporaries" part of its frame) any caller-saves registers whose values are still needed
2. puts arguments into the build area at the top of the current frame
3. places the static link (if any) in register ecx
4. executes a call instruction

The caller-saves registers consist of eax, edx, and ecx. Step 1 is skipped if none of these contain a value that will be needed later. Step 2 is skipped if the subroutine has no parameters. Step 3 is skipped if the language has no nested subroutines, or if the called routine is declared at the outermost nesting level. The call instruction pushes the return address and jumps to the subroutine.

In its prologue, the callee

1. pushes the fp onto the stack (if the current routine uses the fp), implicitly decrementing the sp by 4 (one word).
2. copies the sp into the fp if necessary, thereby establishing a frame pointer for the current routine
3. pushes any callee-saves registers whose values may be overwritten by the current routine
4. pushes the static link (ecx) if the language has nested subroutines and this is not a leaf
5. subtracts the remainder of the frame size from the sp

The callee-saves registers are ebx, esi, edi, and, for routines that don't need a frame pointer, ebp. For routines that do need a frame pointer, registers esp and ebp (the sp and fp, respectively) are saved by Steps 1 and 2. The instructions for some of these steps may be replaced with equivalent sequences by the compiler's code improver, and may be mixed into the rest of the subroutine by the instruction scheduler. In particular, if the value subtracted from the sp in Step 5 is made large enough to accommodate the callee-saves registers, then the pushes in Steps 3 and 4 may be moved after Step 5 and replaced with fp- or sp-relative stores.

In its epilogue, the callee

1. sets the return value
2. restores any callee-saved registers
3. copies the fp into the sp, or subtracts a constant from the sp, as appropriate, thereby deallocating the frame
4. pops the fp, if any, off the stack
5. returns

Steps 3 and 4 may be effected on the x86 by a single leave instruction. As in the previous case study, the caller moves the return value, if it is in a register, to wherever it is needed. It restores any caller-saves registers lazily over time. ■

EXAMPLE 9.72
Subroutine closure trampoline

Because Ada allows subroutines to nest (and Ada 2005 allows arbitrary subroutines to be passed as parameters), a subroutine *S* that is passed as a parameter from *P* to *Q* must be represented by a closure, as described in Section 3.6.1. In many compilers the closure is a data structure containing the address of *S* and the static link that should be used when *S* is called. In gcc, however, the closure contains an *x86 code sequence* known as a *trampoline*—typically a pair of instructions to load ecx with the appropriate static link and then jump to the beginning of *S*. The trampoline resides in the "local variables and temporaries" section of *P*'s activation record. Its address is passed to *Q*. Rather than "interpret" the closure at run time, *Q* actually calls it. One advantage of this mechanism is its interoperability across programming languages: C functions passed as parameters are simply code addresses. In fact, if *S* is declared at the outermost level of lexical nesting, then gcc can pass an ordinary code address even when compiling Ada source; in this case no trampoline is required. ■

x86-64 As noted in Section C-5.4.5, the x86-64 has 16 integer registers instead of only 8. AMD, which developed the ISA for the wider architecture, suggests a calling sequence that makes more use of registers (and less of the stack), in a manner reminiscent of Arm (Example C-9.69) and other RISC machines. The GNU compiler generally conforms to AMD's suggestions.

Figure C-9.10 shows a stack frame for the x86-64. The first six integer arguments are passed in registers rdi, rsi, rdx, rcx, r8, and r9, in that order. The static link, when needed, is passed in r10 (not rcx). Registers rbx and r12–r15 are callee saves; rax, r10, r11, and the argument registers are caller-saves. Integer function values are returned in rax and (if needed) rdx. The first eight floating-point arguments are passed in XMM/SSE registers xmm0–xmm7 (the legacy x87 registers are for the most part ignored). Additional floating-point arguments are passed on the stack. Floating-point function values are returned in xmm0 and (if needed) xmm1. The stack is always 16-byte aligned at the time of a call.

EXAMPLE 9.73

The x86-64 red zone

Perhaps the most interesting difference between the x86-32 and x86-64 conventions is AMD's specification of a "red zone" beyond the sp. Where the last used word on the stack is guaranteed on x86-32 to be at an address no lower than the sp, on x86-64 it can be up to 128 bytes beyond this point—in effect, the sp protects not only the data at higher addresses (below it in the stack), but up to 128 bytes of additional data as well. Signal handlers and other system software are required to respect this convention. As a result, leaf routines that need a stack frame smaller than 128 bytes need not update the sp. For frequent calls to very small routines, the two-instruction savings in per-call bookkeeping can be significant. ■

✓ CHECK YOUR UNDERSTANDING

51. For each of our three case studies, explain which aspects of the calling sequence and stack layout are dictated by the hardware, and which are a matter of software convention.

52. Why don't LLVM and gcc restore caller-saves registers immediately after a call?

53. What is a subroutine closure *trampoline*? How does it differ from the usual implementation of a closure described in Section 3.6.1? What are the comparative advantages of the two alternatives?

DESIGN & IMPLEMENTATION

9.11 Executing code in the stack

A disadvantage of trampoline-based closures is the need to execute code in the stack. Many machines and operating systems disallow such execution, for at least two important reasons. First, as noted in Section C-5.1, modern microprocessors typically have separate instruction and data caches, for fast concurrent access. Allowing a process to write and execute the same region of memory means that these caches must be kept mutually consistent (coherent), a task that introduces significant hardware complexity (on some machines it requires execution of a special hardware instruction). Second, many computer security breaches involve a *code injection* attack, in which an intruder exploits software vulnerabilities (e.g., the lack of array bounds checking in C) to write instructions into the stack, and to overwrite the saved return address so that execution will jump into that code when the current subroutine returns. Such an attack is possible only on machines in which writable data are also executable. When compiling code for use on modern systems, gcc embeds a call to a library routine that reverses the system default and re-enables stack execution prior to using a trampoline.

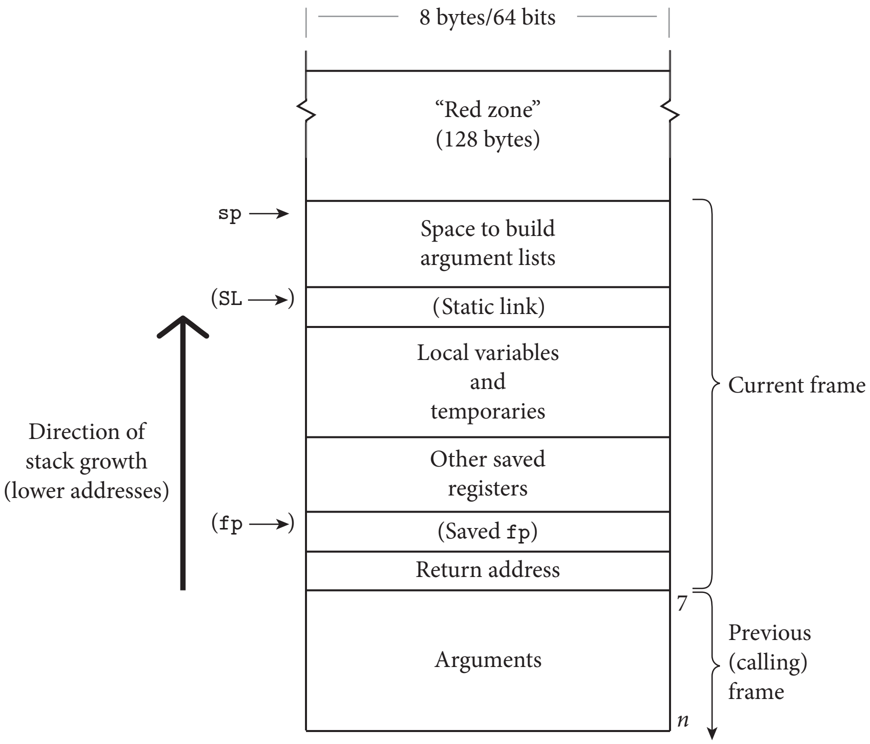


Figure 9.10 Layout of the subroutine call stack for the GNU Compiler Collection (gcc) on 64-bit x86. Conventions differ from those of Figure C-9.9 in three principal ways: (1) most data are 64 bits wide; (2) the first 6 integer arguments are passed in registers rather than on the stack; (3) leaf routines are permitted to use up to 128 bytes of space beyond the top of the stack, without updating the sp.

54. Explain the circumstances under which a subroutine needs a frame pointer (i.e., under which access via displacement addressing from the stack pointer will not suffice).

55. Under what circumstances must an argument that was passed in a register also be saved into the stack?

56. What is the purpose of the "red zone" on x86-64?

9 Subroutines and Control Abstraction

9.2.3 Register Windows

EXAMPLE 9.74

Register windows on the SPARC

As an alternative to saving and restoring registers on subroutine calls and returns, the original Berkeley RISC machines [PD80, Pat85] incorporated a hardware mechanism known as *register windows*. The basic idea is to provide a very large set of physical registers, most of which are organized as a collection of overlapping windows (Figure C-9.11). A few register names (`r0`–`r7` in the figure) always refer to the same locations, but the rest (`r8`–`r31` in the figure) are interpreted relative to the currently active window. On a subroutine call, the hardware moves to a different window. To facilitate the passing of parameters, the old and new windows overlap: the top few registers in the caller's window (`r24`–`r31` in the figure) are the same as the bottom few registers in the callee's window (`r8`–`r15` in the figure). On a machine with register windows, the compiler places values of use only within the current subroutine in the middle part of the window. It copies values to the upper part of the window to pass them to a called routine, within which they are read from the lower part of the window. ■

Since the number of physical windows is fixed, a long chain of subroutine calls can cause the hardware to run off the end of the register set, resulting in a "window overflow" interrupt that drops the processor into the operating system. The interrupt handler then treats the set of available windows as a circular buffer. It copies the contents of one or more windows to memory and then resumes execution. Later, a "window underflow" interrupt will occur when control attempts to return into a window whose contents have been written to memory. Again the operating system recovers, by restoring the saved registers and resuming execution. In practice, eight windows appear to suffice to make overflow and underflow relatively rare in typical programs.

Register windows have been used in several RISC processors, but only one of these, the SPARC, is commercially significant today. The Intel IA-64 (Itanium), introduced shortly after the turn of the century, also uses register windows, though it is not a RISC machine. The advantage of windows, of course, is that they reduce

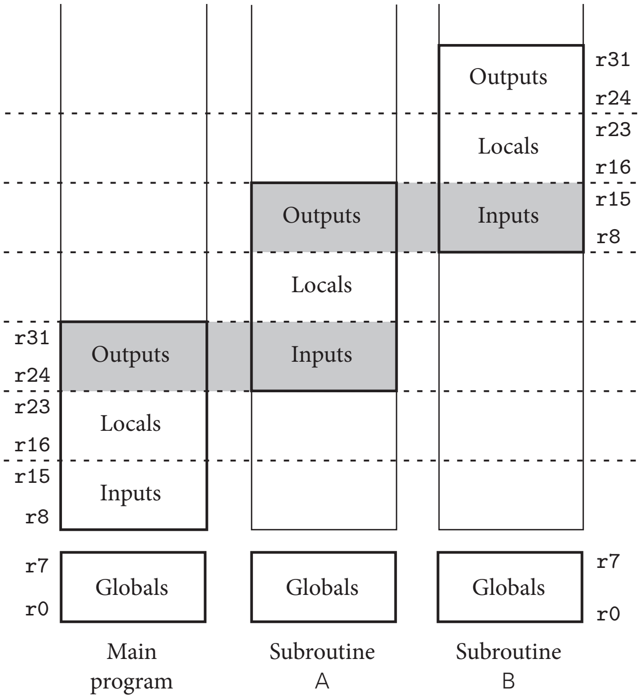


Figure 9.11 **Register windows.** When the main program calls subroutine A, and again when A calls B, register names `r0`–`r7` continue to refer to the same locations, but register names `r8`–`r31` are changed to refer to a new, overlapping window. High-numbered registers in the caller share locations with low-numbered registers in the callee.

the number of loads and stores required for the typical subroutine call. At the same time, register windows significantly increase the amount of state associated with the currently running program. When the operating system decides to give the processor to a different application for a while (something that most systems do many times per second), it must save all this state to memory, or arrange for the processor to trap back into the OS if the new process attempts to access an unsaved window. Worse, while register windows nicely capture the referencing environment of a single thread of control, they do not work well for languages that need more than one referencing environment (execution context). Several language features, including continuations (Section 6.2.2), iterators (Section 6.5.3), and coroutines (Section 9.5), are difficult to implement on a machine with register windows, because they require that we save and restore not only the visible registers, but those in other windows as well, when switching between contexts. It is unclear whether the reduction in subroutine call overhead outweighs the extra cost of context switches for typical application workloads, particularly given that loads and stores for parameters are almost always cache hits.

✓ CHECK YOUR UNDERSTANDING

57. What are *register windows*? What purpose do they serve?
58. Which commercial instruction sets include register windows?
59. Explain the concepts of register window *overflow* and *underflow*.
60. Why are register windows a potential problem for multithreaded programs?

9 Subroutines and Control Abstraction

9.3.2 Call by Name

Call by name implements the normal-order argument evaluation described in Section 6.6.2. A call-by-name parameter is reevaluated in the caller's referencing environment every time it is used. The effect is as if the called routine had been textually expanded at the point of call, with the actual parameter (which may be a complicated expression) replacing every occurrence of the formal parameter. To avoid the usual problems with macro parameters, the "expansion" is defined to include parentheses around the replaced parameter wherever syntactically valid, and to make "suitable systematic changes" to the names of any formal parameters or local identifiers that share the same name, so that their meanings never conflict [NBB$^+$63, p. 12]. Call by name was the default in Algol 60; call by value was available as an alternative. In Simula call by value was the default; call by name was the alternative.

To implement call by name, Algol 60 implementations passed a hidden subroutine that evaluated the actual parameter in the caller's referencing environment. Such a hidden routine is usually called a *thunk*.[1] In most cases thunks are trivial. If an actual parameter is a variable name, for example, the thunk simply reads the variable from memory. In some cases, however, a thunk can be elaborate. Perhaps the most famous occurs in what is known as *Jensen's device*, named after Jørn Jensen [Rut67]. The idea is to pass to a subroutine both a built-up expression and one or more of the variables used in the expression. Then by changing the values of the individual variable(s), the called routine can deliberately and systematically change the value of the built-up expression. This device can be used, for example, to write a summation routine:

EXAMPLE 9.75
Jensen's device

[1] In general, a thunk is a procedure of zero arguments used to delay evaluation of an expression. Other examples of thunks can be seen in the `delay` mechanism of Example 6.88 and the `promise` constructor of Exercise 11.18.

```
real procedure sum(expr, i, low, high);
    value low, high;
        comment low and high are passed by value;
        comment expr and i are passed by name;
    real expr;
    integer i, low, high;
begin
    real rtn;
    rtn := 0;
    for i := low step 1 until high do
        rtn := rtn + expr;
        comment the value of expr depends on the value of i;
    sum := rtn
end sum
```

Now to evaluate the sum

$$y = \sum_{1 \leq x \leq 10} 3x^2 - 5x + 2$$

we can simply say

```
y := sum(3*x*x - 5*x + 2, x, 1, 10);
```

Label Parameters

Both Algol 60 and Algol 68 allowed a label to be passed as a parameter. If a called routine performed a `goto` to such a label, control would usually need to escape the local context, unwinding the subroutine call stack as it did so. Details of the unwinding operation would depend on the location of the label. For each intervening scope, the `goto` would have to restore saved registers, deallocate the

DESIGN & IMPLEMENTATION

9.12 Call by name

In practice, most uses of call by name in Algol 60 and Simula programs served to allow a subroutine to change the value of an actual parameter; neither language offered call by reference. Unfortunately, call by name is significantly more expensive than call by reference: it requires the invocation of a thunk (as opposed to a simple indirection) on every use of a formal parameter. Call by name is also prone to subtle program bugs when a change to a variable in a surrounding scope unintentionally alters the value of a formal parameter. (Call by reference suffers from a milder form of this problem, as discussed in Example 3.20.) Such deliberate subtleties as Jensen's device are comparatively rare, and can be imitated in other languages through the use of formal subroutines. Call by name was dropped in Algol 68, in favor of call by reference.

stack frame, and perform any other operations normally handled by epilogue code. To implement label parameters, Algol implementations typically passed a thunk that performed the appropriate operations for the given label. Note that the target label would generally need to lie in some surrounding scope, where it was visible to the caller under static scoping rules.

Label parameters were usually used to handle *exceptional conditions*—conditions that prevent a subroutine from performing its usual operation, and that cannot be handled in the local context. Instead of returning, an Algol routine that encountered a problem (e.g., invalid input) could perform a `goto` to a label parameter, on the assumption that the label referred to code that would perform some remedial operation, or print an appropriate error message. In more recent languages, label parameters have been replaced by more structured exception handling mechanisms, as discussed in Section 9.4.

✓ CHECK YOUR UNDERSTANDING

61. What is *call by name*? What language first provided it? Why isn't it used by the language's descendants?

62. What is *call by need*? How does it differ from call by name? What modern languages use it?

63. How does a subroutine with call-by-name parameters differ from a macro?

64. What is a *thunk*? What is it used for?

65. What is *Jensen's device*?

DESIGN & IMPLEMENTATION

9.13 Call by need

Functional languages like Miranda and Haskell typically pass parameters using a *memoizing* implementation of normal-order evaluation, as described in Section 6.6.2. This *lazy* implementation is sometimes called *call by need*. Memoization calculates and records the value of a parameter the first time it is needed, and uses the recorded value thereafter. In the absence of side effects, call by need is indistinguishable from call by name. It avoids the expense of repeated evaluation, but precludes the use of techniques like Jensen's device in languages that *do* have side effects. Among imperative languages, call by need appears in the scripting language R, where it serves to avoid the expense of evaluating (even once) any complex arguments that are not actually needed.

9 Subroutines and Control Abstraction

9.5.3 Implementation of Iterators

EXAMPLE 9.76

Coroutine-based iterator invocation

Consider the following `for` loop from Example 6.66:

```
for i in range(first, last, step):
    ...
```

Using coroutines, a compiler might translate this as

```
iter := new from_to_by(first, last, step, i, done, current_coroutine)
while not done do
    . . .
    transfer(iter)
destroy(iter)
```

After the loop completes, the implementation can reclaim the space consumed by iter. ■

EXAMPLE 9.77

Coroutine-based iterator implementation

The definition of from_to_by itself is quite straightforward:

```
coroutine from_to_by(from_val, to_val, by_amt : int;
                    ref i : int; ref done : bool; caller : coroutine)
    i := from_val
    if by_amt > 0 then
        done := from_val ≥ to_val
        detach
        loop
            i +:= by_amt
            done := i ≥ to_val
            transfer(caller)      –– yield i
```

```
else
    done := from_val ≤ to_val
    detach
    loop
        i +:= by_amt
        done := i ≤ to_val
        transfer(caller)      –– yield i
```

Parameters i and done are passed by reference so that the iterator can modify them in the caller's context. The caller's identity is passed as a final argument so that the iterator can tell which coroutine to resume when it has computed the next loop index. Because the caller is named explicitly, it is easy for iterators to nest, as in Figure 6.5.

Single-Stack Implementation

While coroutines *suffice* for the implementation of iterators, they are not *necessary*. A simpler, single-stack implementation is also possible. Because a given iterator (e.g., an instance of `from_to_by`) is always resumed at the same place in the code (between iterations of a given `for` loop), we can be sure that the subroutine call stack will always contain the same frames whenever the iterator runs. Moreover, since `yield` statements can appear only in the main body of the iterator (never in nested routines), we can be sure that the stack will always contain the same frames whenever the iterator transfers back to its caller. These two facts imply that we can place the frame of the iterator directly on top of the frame of its caller in a single central stack.

When an iterator is created, its frame is pushed on the stack. When it yields a value, control returns to the `for` loop, but the iterator's frame is left on the stack. If the body of the loop makes any subroutine calls, the frames for those calls will be allocated beyond the frame of the iterator. Since control must return to the loop before the iterator resumes, we know that such frames will be gone again before the iterator has a chance to see them: if it needs to call subroutines itself, the stack above it will be clear. Likewise, if the iterator calls any subroutines, they will return (popping their frames from the stack) before the `for` loop runs again. Nested iterators present no special problems (see Exercise C-9.37).

Data Structure Implementation

Compilers for C# 2.0 employ yet another implementation of iterators. Like Java, C# 1.1 provided iterator objects. Each such object implements the `IEnumerator` interface, which provides `MoveNext` and `Current` methods. Typically an iterator is obtained by calling the `GetEnumerator` method of an object (a container) that implements the `IEnumerable` interface:

EXAMPLE 9.78
Iterator usage in C#

```
for (IEnumerator i = myTree.GetEnumerator(); i.MoveNext();) {
    object o = i.Current;
    Console.WriteLine(o.ToString());
}
```

EXAMPLE 9.79
Implementation of C# iterators

C# 2.0 provides true iterators as an extension of iterator objects. The programmer simply declares a method that contains one or more `yield return` statements, and whose return type is `IEnumerator` or `IEnumerable`. Here is an example of the latter:

```
static IEnumerable FromToBy (int fromVal, int toVal, int byAmt)
{
    if (byAmt >= 0) {
        for (int i = fromVal; i <= toVal; i += byAmt) {
            yield return i;
        }
    } else {
        for (int i = fromVal; i >= toVal; i += byAmt) {
            yield return i;
        }
    }
}
```

The compiler automatically transforms this code into a hidden class with a `GetEnumerator` method, along the lines of Figure C-9.12. Within this code, an explicit state variable keeps track of the "program counter" of the last `yield` statement. In addition, local variable `i` of the true iterator becomes a data member of the `FromToByImpl` class, leaving the iterator with no need for a stack frame across iterations of the loop. In a quite literal sense, the compiler transforms each true iterator into an iterator object. ■

Recursive iterators present no particular difficulties: a nested iterator is allocated on demand when the outer iterator enters a `foreach` loop, and is referred to by a reference in that outer iterator. The details are deferred to Exercise C-9.38. Because iterator objects are allocated from the heap, the C# implementation of true iterators may be somewhat slower than the stack-based implementation of the previous subsection.

✓ CHECK YOUR UNDERSTANDING

66. Describe the "obvious" implementation of iterators using coroutines.

67. Explain how the state of multiple active iterators can be maintained in a single stack.

68. Describe the transformation used by C# compilers to turn a true iterator into an iterator object.

```
static IEnumerable FromToBy(int fromVal, int toVal, int byAmt) {
    return new FromToByImpl(fromVal, toVal, byAmt);
}
class FromToByImpl : IEnumerator, IEnumerable {
    enum State {starting, goingUp, goingDown, done}
    int i, tv, ba;
    State s;

    public FromToByImpl(int fromVal, int toVal, int byAmt) {
        i = fromVal; tv = toVal; ba = byAmt; s = State.starting;
    }
    public IEnumerator GetEnumerator() {
        return this;
    }
    public object Current {
        get { return i; }
    }
    public bool MoveNext() {
        switch (s) {
            case State.starting :
                if (ba >= 0) {
                    if (i <= tv) { s = State.goingUp; return true; }
                    else { s = State.done; return false; }
                } else {
                    if (i >= tv) { s = State.goingDown; return true; }
                    else { s = State.done; return false; }
                }
            case State.goingUp :
                i += ba;
                if (i <= tv) return true;
                else { s = State.done; return false; }
            case State.goingDown :
                i += ba;
                if (i >= tv) return true;
                else { s = State.done; return false; }
            default: // for completeness
            case State.done : return false;
        }
    }
    public void Reset() {
        s = State.starting;
    }
}
```

Figure 9.12 **Iterator object equivalent of a true iterator in C#.** This handwritten code corresponds to Example C-9.79. It represents, at the source level, what the compiler creates at the level of intermediate code: a state machine that tracks the program counter of the original iterator, with a starting state, an ending state, and one state for each `yield return` statement. The arms of the `switch` statement capture the code paths in the original iterator that move from one state to the next.

9 Subroutines and Control Abstraction

9.5.4 Discrete Event Simulation

EXAMPLE 9.80

Sequential simulation of a complex physical system

Suppose that we wish to experiment with the flow of traffic in a city. A computerized traffic model, if it captures the real world with sufficient accuracy, will allow us to predict the effects of construction projects, accidents, increased traffic due to new development, or changes to the layout of streets. It is difficult (though certainly not impossible) to write such a simulation in a conventional sequential language. We would probably represent each interesting object (automobile, intersection, street segment, etc.) with a data structure. Our main program would then look something like this:

```
while current_time < end_of_simulation
    calculate next time t at which an interesting interaction will occur
    current_time := t
    update state of objects to reflect the interaction
    record desired statistics
print collected statistics
```

The problem with this approach lies in determining which objects will interact next, and in remembering their state from one interaction to the next. It is in some sense unnatural to represent active objects such as cars with passive data structures, and to make time the active entity in the program. An arguably more attractive approach is to represent each active object with a coroutine, and to let each object keep track of its own state.

EXAMPLE 9.81

Initialization of a coroutine-based traffic simulation

If each active object can tell when it will next do something interesting, then we can determine which objects will interact next by keeping the currently inactive coroutines in a priority queue, ordered by the time of their next event. We might begin a one-day traffic simulation by creating a coroutine for each trip to be taken by a car that day, and inserting each coroutine into the priority queue with a "wakeup" time indicating when the trip is to begin:

```
coroutine trip(. . . )
. . .
for each trip t
    p := new trip(. . . )
    schedule(p, t.start_time)
```

EXAMPLE 9.82

Traversing a street segment in the traffic simulation

Let us assume that we think of street segments as passive, and represent them with data structures. At any given moment, we can model a segment by the number of cars that it is carrying in each direction. This number in turn will affect the speed at which the cars can safely travel. Whenever it awakens, the coroutine representing a trip examines the next street segment over which it needs to travel. Based on the current load on that segment, it calculates how much time it will take to traverse it, and schedules itself to awaken again at an appropriate point in the future:

```
coroutine trip(origin, destination : location)
    plan a route from origin to destination
    detach
    for each segment of the route
        calculate time i to reach the end of the segment
        schedule(current_coroutine, current_time + i)
```

EXAMPLE 9.83

Scheduling a coroutine for future execution

The `schedule` operation is easily built on top of transfer:

```
schedule(p : coroutine; t : time)
    –– p may be self or other
    insert (p, t) in priority queue
    if p = current_coroutine      –– self
        extract earliest pair (q, s) from priority queue
        current_time := s
        transfer(q)
```

EXAMPLE 9.84

Queueing cars at a traffic light

In some cases, it may be difficult to determine when to reschedule a given object. Suppose, for example, that we wish to more accurately model the effects of traffic signals at intersections. We might represent each traffic signal with a data structure that records the waiting cars in each direction, and a coroutine that lets cars through as the signal changes color:

```
record controlled_intersection =
    EW_cars, NS_cars : queue of trip
    const per_car_lag_time : time
        –– how long it takes a car to start after its predecessor does
    coroutine signal(EW_duration, NS_duration : time)
        detach
        loop
            change_time := current_time + EW_duration
            while current_time < change_time
                if EW_cars not empty
                    schedule(dequeue(EW_cars), current_time)
                schedule(current_coroutine, current_time + per_car_lag_time)
            change_time := current_time + NS_duration
            while current_time < change_time
                if NS_cars not empty
                    schedule(NS_cars.dequeue(), current_time)
                schedule(current_coroutine, current_time + per_car_lag_time)
```

EXAMPLE 9.85
Waiting at a light

When it reaches the end of a street segment that is controlled by a traffic signal, a trip need not calculate how long it will take to get through the intersection. Rather, it enters itself into the appropriate queue of waiting cars and "goes to sleep," knowing that the `signal` coroutine will awaken it at some point in the future:

```
coroutine trip(origin, destination : location)
    plan a route from origin to destination
    detach
    for each segment of the route
        calculate time $i$ to reach the end of the segment
        schedule(current_coroutine, current_time + $i$)
        if end of segment has a traffic light
            identify appropriate queue $Q$
            $Q$.enqueue(current_coroutine)
            sleep()
```

EXAMPLE 9.86
Sleeping in anticipation of future execution

Like `schedule`, `sleep` is easily built on top of transfer:

```
sleep()
    extract earliest pair ($q$, $s$) from priority queue
    current_time := $s$
    transfer($q$)
```

The `schedule` operation, in fact, is simply

```
schedule(p : coroutine; t : time)
    insert (p, t) in priority queue
    if p = current_coroutine
        sleep()
```

Obviously this traffic simulation is too simplistic to capture the behavior of cars in a real city, but it illustrates the basic concepts of discrete event simulation. More sophisticated simulations are used in a wide range of application domains, including all branches of engineering, computational biology, physics and cosmology, and even computer design. Multiprocessor simulations (see reference [VF94], for example) are typically divided into a "front end" that simulates the processors and a "back end" that simulates the memory subsystem. Each coroutine in the front end consists of a machine-language interpreter that captures the behavior of one of the system's processing cores. Each coroutine in the back end represents a load or a store instruction. Every time a processor performs a load or store, the front end creates a new coroutine in the back end. Data structures in the back end represent various hardware resources, including caches, buses, network links, message routers, and memory modules. The coroutine for a given load or store checks to see if its location is in the local cache. If not, it must traverse the interconnection network between the processor and memory, competing with other coroutines for access to hardware resources, much as cars in our simple example compete for access to street segments and intersections. The behavior of the back-end system in turn affects the front end, since a processor must wait for a load to complete before it can use the data, and since the rate at which stores can be injected into the back end is limited by the rate at which they propagate to memory.

✓ **CHECK YOUR UNDERSTANDING**

69. Summarize the computational model of discrete event simulation. Explain the significance of the time-based priority queue.

70. When building a discrete event simulation, how does one decide which things to model with coroutines, and which to model with data structures?

71. Are all inactive coroutines guaranteed to be in the priority queue? Explain.

9 Subroutines and Control Abstraction

9.9 Exercises

9.29 Suppose you wish to minimize the size of closures in a language implementation that uses a display to access nonlocal objects. Assuming a language like Pascal or Ada, in which subroutines have limited extent, explain how an appropriate display for a formal subroutine can be calculated when that routine is finally called, starting with only (1) the value of the frame pointer, saved in the closure at the time that the closure was created, (2) the subroutine return addresses found in the stack at the time the formal subroutine is finally called, and (3) static tables created by the compiler. How costly is your scheme?

9.30 Elaborate on the reasons why even parameters passed in registers may sometimes need to have locations in the stack. Consider all the cases in which it may not suffice to keep a parameter in a register throughout its lifetime.

9.31 Most versions of the C library include a function, `alloca`, that dynamically allocates space within the current stack frame.[2] It has two advantages over the usual `malloc`, which allocates space in the heap: it's usually very fast, and the space it allocates is reclaimed automatically when the current subroutine returns. Assuming the programmer *wants* deallocation to happen then, it's convenient to be able to skip the explicit `free` operations. How might you implement `alloca` in conjunction with the calling conventions of our various case studies?

9.32 Explain how to extend the conventions of Figure C-9.9 and Section C-9.2.2 to accommodate arrays whose bounds are not known until elaboration time (as discussed in Section 8.2.2). What ramifications does this have for the use of separate stack and frame pointers?

[2] Unfortunately, `alloca` is not POSIX compliant, and implementations vary greatly in their semantics and even in details of the interface. Portable programs are wise to avoid this routine.

9.33 In all three of our case studies, stack-based arguments were placed into the argument build area in "reverse" order, with the lowest-numbered argument at the top. Explain why this is important. (Hint: Consider subroutines with variable numbers of arguments, as discussed in Section 9.3.3.)

9.34 How would you implement nested subroutines as parameters on a machine that doesn't let you execute code in the stack? Can you pass a simple code address, or do you require that closures be interpreted at run time?

9.35 If you have read the rest of Chapter 9, you may have noticed that the term "trampoline" is also used in conjunction with the implementation of signal handlers (Section 9.6.1). What is the connection (if any) between these uses of the term?

9.36 Explain how you might implement `setjmp` and `longjmp` on a SPARC.

9.37 Following the code in Figure 6.5, and assuming a single-stack implementation of iterators, trace the contents of the stack during the execution of a `for` loop that iterates over all nodes of a complete, 3-level (6-node) binary tree.

9.38 Build a preorder iterator for binary trees in Java, C#, or Python. Do not use a true iterator or an explicit stack of tree nodes. Rather, create nested iterator objects on demand, linking them together as a C# compiler might if it were building the iterator object equivalent of a true preorder iterator.

9.39 One source of inaccuracy in the traffic simulation of Section C-9.5.4 has to do with the timing at traffic signals. If a signal is currently green in the `EW` direction, but the queue of waiting cars is empty, the `signal` coroutine will go to sleep until `current_time + EW_duration`. If a car arrives before the coroutine wakes up again, it will needlessly wait. Discuss how you might remedy this problem.

9 Subroutines and Control Abstraction

9.10 Explorations

9.53 Read the Arm calling sequence standard for 64-bit (v8) code. Compare and contrast to the conventions of Section C-9.2.2. Pay particular attention to the lists of caller- and callee-saves registers, and to the registers used to pass arguments. Speculate as to reasons for the differences.

9.54 Research the full range of hardware support for subroutines on the x86, including all variants of `call`. Note that the `leave` instruction is sometimes generated by modern compilers, but others, including `enter`, `pushad`, `popad`, `pushfd`, and `popfd`, usually are not. In addition, the optional argument of `ret` is almost never used, and `push` and `pop` are used sparingly. Discuss the technological trends that have made this machinery obsolete.

9.55 As an example of hard-core CISC design, research the subroutine calling conventions of the Digital VAX. Be sure to describe the behavior of the `calls` instruction in detail.

9.56 Study the implementation of a user-level thread management package written for the SPARC. How does it manage register windows?

9.57 Learn how parameter passing is implemented in the Glasgow Haskell compiler. How expensive is its call-by-need–based lazy evaluation?

9.58 Learn about the Time Warp system for discrete event simulation, developed by David Jefferson and colleagues [JBW+87]. Discuss its relationship to both the classic discrete event simulation of Section C-9.5.4 and the speculative parallelism of mechanisms like transactional memory (to be discussed in Section 13.4.5).

10 Object Orientation

10.6 True Multiple Inheritance

EXAMPLE 10.56
Deriving from two base classes (reprise)

Recall our administrative computing example in C++:

```
class student : public person, public system_user { ...
```

To implement multiple inheritance, we must be able to generate both a "`person` view" and a "`system_user` view" of a `student` object on demand, for example when assigning a reference to a `student` object into a `person` or `system_user` variable. For one of the base classes (`person`, say) we can do the same thing we did with single inheritance: let the data members of that base class lie at the beginning of the representation of the derived class, and let the virtual methods of that base class lie at the beginning of the vtable. Then when we assign a reference to a `student` object into a `person` variable, code that manipulates the `person` variable will just use a prefix of the data members and the vtable. ■

EXAMPLE 10.57
(Nonrepeated) multiple inheritance

For the other base class (`system_user`), things get more complicated: we can't put *both* base classes at the beginning of the derived class. One possible solution is shown in Figure C-10.8. It is based loosely on the implementation described by Ellis and Stroustrup [ES90, Chap. 10]. Because the `system_user` fields of a `student` follow the `person` fields, the assignment of a reference to a `student` object into a variable of type `system_user*` requires that we adjust our "view" by adding the compile-time constant offset d.

The vtable for a `student` is broken into two parts. The first part lists the virtual methods of the derived class and the first base class (`person`). The second part lists the virtual methods of the second base class. (We have already introduced a method, `print_mailing_label`, defined in class `person`. We may similarly imagine that `system_user` defines a virtual method `print_stats` that is supposed to dump account statistics to standard output.) Generalization to three or more base classes is straightforward; see Exercise C-10.23.

Every data member of a `student` object has a compile-time-constant offset from the beginning of the object. Likewise, every virtual method has a compile-time-constant offset from the beginning of one of the parts of the vtable. The address of

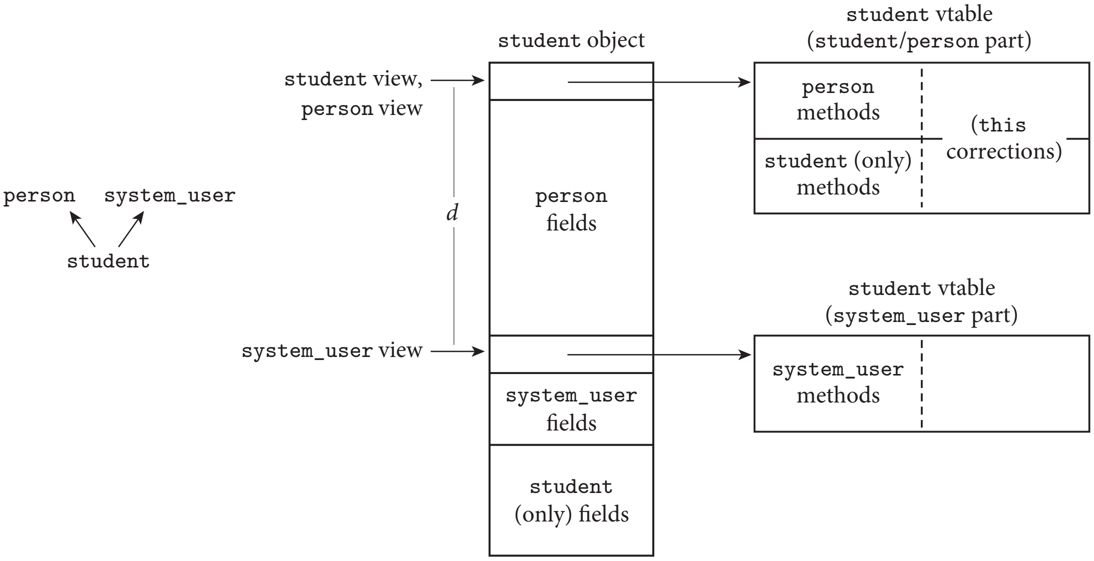


Figure 10.8 Implementation of (nonrepeated) multiple inheritance. The size *d* of the `person` portion of the object is a compile-time constant. We access the `system_user` portion of the vtable by adding *d* to the address of a `student` object before indirecting. Likewise, we create a `system_user` view of a `student` object by adding *d* to the object's address. Each vtable entry consists of both a method address and a "`this` correction" value equal to the signed distance between the view through which the vtable was accessed and the view of the class in which the method was defined.

the `person/student` portion of the vtable is stored in the beginning of the object. The address of the `system_user` portion of the vtable is stored at offset *d*. Note that both parts of the vtable are specific to class `student`. In particular, the `system_user` part of the vtable is *not* shared by objects of class `system_user`, because the contents of the tables will be different if `student` has overridden any of `system_user`'s virtual methods.

EXAMPLE 10.58

Method invocation with multiple inheritance

To call the virtual method `print_mailing_label`, originally defined in `person`, we can use a code sequence similar to the one shown in Section 10.4.2 for single inheritance. To call a virtual method originally defined in `system_user`, we must first add the offset *d* to our object's address, in order to find the address of the `system_user` portion of the vtable. Then we can index into this `system_user` vtable to find the address of the appropriate method to call. But we are left with one final problem: what is the appropriate value of `this` to pass to the method?

As a concrete example, suppose that `student` does not override `print_stats` (though it certainly could). If our object is of class `student`, we should pass a `system_user` view of it to `print_stats`: the address of the object, plus *d*. If, however, our object is of some class (`transfer_student`, perhaps) that does override `print_stats`, then we should pass a `transfer_student` view to `print_stats`. If we are accessing our object through a variable (a reference or a pointer) whose methods are dynamically bound, then we can't tell at compile time which one of these cases applies. Worse yet, we may not even know how to generate a `transfer_student` view if we have to: class `transfer_student` may not have been invented when this part of our code was compiled, so we certainly don't know how far into it the `system_user` fields appear! ■

EXAMPLE 10.59
`this` correction

A common solution is for each vtable entry to consist of a *pair* of fields. One is the address of the method's code; the other is a "`this` correction" value, to be added to the view through which we found the vtable. Returning to Figure C-10.8, the "`this` correction" field of the vtable entry for `print_stats` would contain $-d$ if `print_stats` was overridden by `student`, and zero otherwise. In the `system_user` part of the vtable for the (yet to be written) class `transfer_student`, the "`this` correction" field might contain some other value $-e$. In general, the "`this` correction" is the distance between the view of the class in which the method was *declared* (and through which we accessed the vtable) and the view of the class in which the method was *defined* (and which will therefore be expected by the subroutine's implementation).

If variable `my_student` contains a reference to (a student view of) some object at run time, and if `print_stats` is the third virtual method of `system_user`, then the code to call `my_student.print_stats` would look something like this:

```
r1 := my_student                  –– student view of object
r1 := r1 + d                      –– system_user view of object
r2 := *r1                         –– address of appropriate vtable
r3 := *(r2 + (3−1) × 8)           –– method address
r2 := *(r2 + (3−1) × 8 + 4)       –– this correction
r1 := r1 + r2                     –– this
call *r3
```

Here we have assumed that both method addresses and `this` corrections are four bytes long, that `this` is to be passed in `r1`, and that there are no other arguments. On a typical machine this code is three instructions (including one memory access) longer than the code required with single inheritance, and five instructions (including three memory accesses) longer than a call to a statically identified method. ■

DESIGN & IMPLEMENTATION

10.9 The cost of multiple inheritance

The implementation we have described for multiple inheritance, using `this` corrections in vtables, has the unfortunate property of increasing the overhead of all virtual method invocations, even in programs that do not make use of multiple inheritance. This sort of mandatory overhead is something that language designers (and the designers of systems languages in particular) generally try to avoid; as a matter of principle, complex special cases should not reduce the efficiency of the simpler common case. Fortunately, there are other implementations of multiple inheritance (see Exercise C-10.28) in which the cost of modifying `this` is paid only when the correction is nonzero.

10.6.1 Semantic Ambiguities

EXAMPLE 10.60
Methods found in more than one base class

In addition to implementation complexities (only some of which we have discussed so far), multiple inheritance introduces potential semantic problems. Suppose that both system_user and person define a print_stats method. If we have a variable s of type student* and we call s->print_stats, which version of the method should we get? In CLOS and Python, we get the version from the base class that appeared first in the derived class's header. In Eiffel, we get a static semantic error if we try to define a derived class with such an ambiguity. In C++, we can define the derived class, but we get a static semantic error if we attempt to use a member whose name is ambiguous. To resolve the ambiguity, we can use the feature renaming mechanism in Eiffel to give different names to the inherited methods. In C++ we must redefine the conflicting method explicitly:

```
void student::print_stats() {
    person::print_stats();
    system_user::print_stats();
}
```

Here we have chosen to call the print_stats routines of both base classes, using the :: scope resolution operator to name them. We could of course have chosen to call just one, or to write our own code from scratch. We could even arrange for access to both routines by giving them new names:

```
void student::print_person_stats() {
    person::print_stats();
}
void student::print_user_stats() {
    system_user::print_stats();
}
```

■

EXAMPLE 10.61
Overriding an ambiguous method

Things are a little messier if either or both of the identically named base class methods are virtual, and we want to override them in the derived class. Following Stroustrup [Str13, Sec. 21.3.3], we can solve the problem by interposing an intermediate class between each base class and the derived class:

```
class person_interface : public person {
public:
    virtual void print_person_stats() = 0;
    void print_stats() { print_person_stats(); }
        // overrides person::print_stats
};
class system_user_interface : public system_user {
public:
    virtual void print_user_stats() = 0;
    void print_stats() { print_user_stats(); }
        // overrides system_user::print_stats
};
```

```
class student : public person_interface, public system_user_interface {
public:
    void print_person_stats() { ...
    void print_user_stats() { ...
    ...
};
```

We leave it as an exercise (C-10.24) to show what happens if we assign a `student` object into a variable `p` of type `person*` and then call `p->print_stats()`. ■

EXAMPLE 10.62

Repeated multiple inheritance

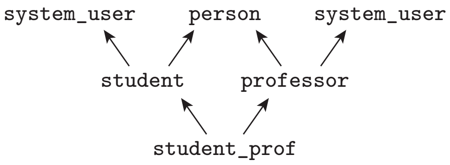
A more serious ambiguity arises when a class *D* inherits from two base classes, *B* and *C*, both of which inherit from some common base class *A*. In this situation, should an object of class *D* contain one instance of the data members of class *A* or two? The answer would seem to be program dependent. For example, suppose that professors, like students, are all given accounts in our administrative computing system. Then, like class `student`, we might want class `professor` to inherit from both `person` and `system_user`:

```
class professor : public person, public system_user { ...
```

But now suppose that some professors take courses on occasion as nonmatriculated students. In this case we might want a new class that supports both sets of operations:

```
class student_prof : public student, public professor { ...
```

Class `student_prof` inherits from `person` and from `system_user` twice, once each through `student` and `professor`. If we think about it, we probably want a `student_prof` to have *one* instance of the data members of class `person`—one name, one university ID number, one mailing address—and *two* instances of the data members of class `system_user`—separate user accounts (with separate user ids, disk quotas, etc.) for the student and professor roles:



The `system_user` case—separate copies from each branch of the inheritance tree—is known as *replicated inheritance*. The `person` case—a single copy from both branches of the tree—is known as *shared* inheritance. Both are forms of *repeated inheritance*. ■

EXAMPLE 10.63

Shared inheritance in C++

Replicated inheritance is the default in C++. Shared inheritance is the default in Eiffel. Shared inheritance can be obtained in C++ by specifying that a base class is `virtual`:

```
class student : public virtual person, public system_user { ...
class professor : public virtual person, public system_user { ...
```

In this case the members of class `person` are shared when inherited over multiple paths, while the members of class `system_user` are replicated. ■

EXAMPLE 10.64

Replicated inheritance in Eiffel

Replicated inheritance of individual features can be obtained in Eiffel by means of renaming:

```
class student inherit person; system_user ...
class professor inherit person; system_user ...

class student_prof
inherit
    student
        rename
            user_id as student_user_id,
            disk_quota as student_disk_quota
        end;
    professor
        rename
            user_id as prof_user_id,
            disk_quota as prof_disk_quota
        end
feature
    ...
end -- class student_prof
```

Features inherited with different final names are replicated; features inherited with the same final name are shared. Multiple inheritance in CLOS is always shared, unless the user interposes interface classes as shown in Example C-10.61 explicitly; there is no other renaming mechanism. ■

10.6.2 Replicated Inheritance

EXAMPLE 10.65

Using replicated inheritance

Replicated inheritance introduces no serious implementation problems beyond those of nonrepeated multiple inheritance. As shown in Figure C-10.9, an object (in this case of class `D`) that inherits a base class (`A`) over two different paths in the inheritance tree has two copies of `A`'s data members in its representation, and a set of entries for the virtual methods of `A` in each of the parts of its vtable. Creation of a `B` view of a `D` object (e.g., when assigning a pointer to a `D` object into a `B*` variable) would not require the execution of any code. Creation of a `C` view (e.g., when assigning into a `C*` variable) would require the addition of offset *d*.

Because of ambiguity, we cannot access `A` members of a `D` object by name. We can access them, however, if we assign a pointer to a `D` object into a `B*` or `C*` variable. Similarly, a pointer to a `D` object cannot be assigned into an `A` pointer directly: there would be no basis on which to choose the `A` for which to create a view. We can, however, perform the assignment through a `B*` or `C*` intermediary:

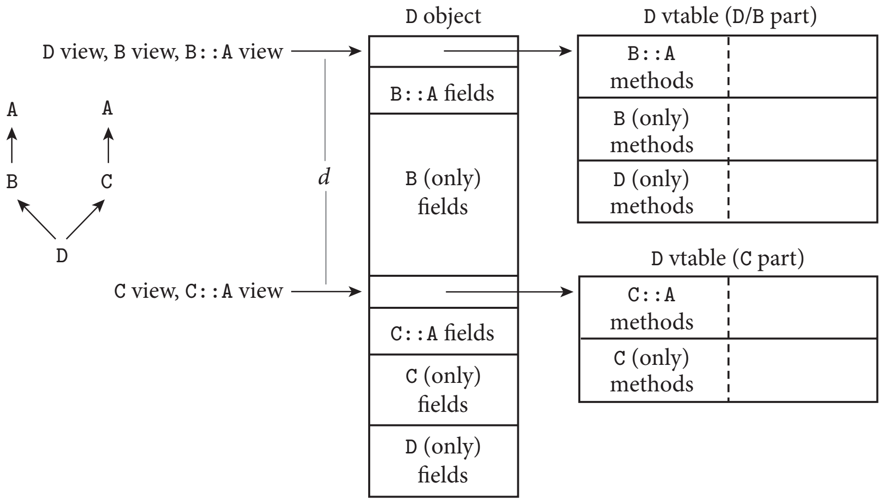


Figure 10.9 **Implementation of replicated multiple inheritance.** Each base class contains a complete copy of class A. As in Figure C-10.8, the vtable for class D is split into two parts, one for each base class, and each vtable entry consists of a ⟨method address, this correction⟩ pair.

```
class A { ...
class B : public A { ...
class C : public A { ...
class D : public B, public C { ...
...
A* a;   B* b;   C* c;   D* d;
a = d;  // error; ambiguous
b = d;  // ok
c = d;  // ok
a = b;  // ok; a := d's B's A
a = c;  // ok; a := d's C's A
```

As described in Example C-10.59, vtable entries will need to consist of ⟨method address, this correction⟩ pairs. ■

10.6.3 Shared Inheritance

EXAMPLE 10.66
Overriding methods with shared inheritance

Shared inheritance introduces a new opportunity for ambiguity and additional implementation complexity. As in the previous subsection, assume that D inherits from B and C, both of which inherit from A. This time, however, assume that A is shared:

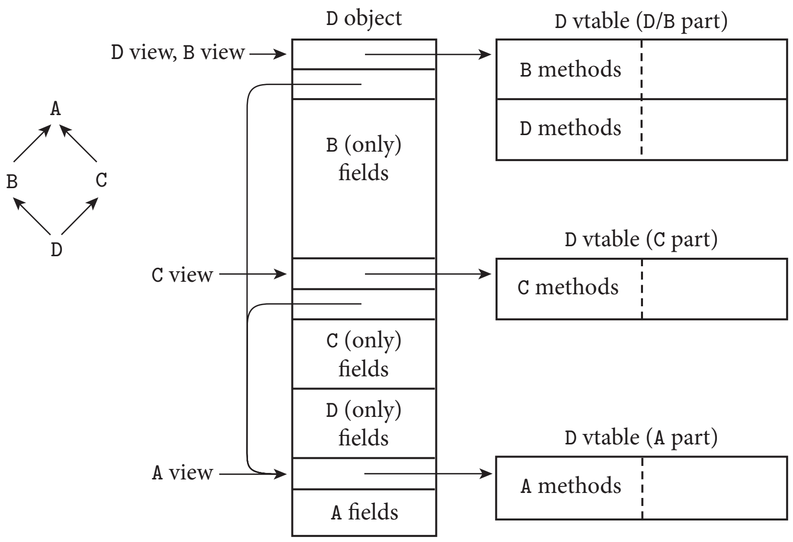


Figure 10.10 **Implementation of shared multiple inheritance.** Objects of class B, C, and D contain the address of their A components at a compile-time constant offset (in this case, immediately after the vtable address). As in Figures C-10.8 and C-10.9, `this` corrections for virtual methods in vtable entries are relative to the view of the class in which the method was declared (i.e., through which the vtable was accessed).

```
class A {
public:
    virtual void f();
    ...
};
class B : public virtual A { ...
class C : public virtual A { ...
class D : public B, public C { ...
```

The new ambiguity arises if B or C overrides method f, declared in A: which version (if any) does D inherit? C++ defines a reference to f to be unambiguous (and therefore valid) if one of the possible definitions *dominates* the others, in the sense that its class is a descendant of the classes of all the other definitions. In our specific example, D can inherit an overridden version of f from either B or C. If both of them override it, however, any attempt to use f from within D's code will be a static semantic error. Eiffel provides comparatively elaborate mechanisms for controlling ambiguity. A class that inherits an overridden method over more than one path can specify the version it wants. Alternatively, through renaming, it can retain access to all versions.

EXAMPLE 10.67
Implementation of shared inheritance

To implement shared inheritance we must recognize that because a single instance of A is a part of both B and C, we cannot make the representations of both B and C contiguous in memory. In Figure C-10.10, in fact, we have chosen to make neither B nor C contiguous. We insist, however, that the representation of every B, C, or D object (and every B, C, or D view of an object of a derived class) contain the address of the A part of the object at a compile-time constant offset from the beginning of the view. To access a data member of A, we first indirect through this address, and then apply the offset of the member within A. To call the *n*th virtual method declared in A, we execute the following code:

```
r1 := my_D_view                  –– original view of object
r1 := *(r1 + 4)                  –– A view
r2 := *r1                        –– address of A part of vtable
r3 := *(r2 + (n−1) × 8)          –– method address
r2 := *(r2 + (n−1) × 8 + 4)      –– this correction
r1 := r1 + r2                    –– this
call *r3
```

This code sequence is the same number of instructions in length as our sequence for nonvirtual base classes (Example C-10.59), but involves one more memory access (to indirect through the A address). The code will work with any D view of any object, including an object of a class derived from D, in which the D and A views might be more widely separated. The constant 4 in the second line assumes 4-byte addresses, with the address of D's A part located immediately after D's initial vtable address. In an object with more than one virtual base class, the address of the part of the object corresponding to each such base would be found at a different offset from the beginning of the object.

The implementation strategy of Figure C-10.10 works in C++ because we always know when a base class is virtual (shared). For data members and virtual methods of nonvirtual base classes, we continue to use the (cheaper) lookup algorithms of Figures C-10.8 and C-10.9. In Eiffel, on the other hand, a feature that is inherited via replication at one level of the class hierarchy may be inherited via sharing later on. As a result, Eiffel requires a somewhat more elaborate implementation strategy (see Exercise C-10.29).

We can avoid the extra level of indirection when accessing virtual methods of virtual base classes in C++ if we are willing to replicate portions of a class's vtable. We explore this option in Exercise C-10.30.

CHECK YOUR UNDERSTANDING

45. Give a few examples of the semantic ambiguities that arise when a class has more than one base class.

46. Explain the distinction between replicated and shared multiple inheritance. When is each desirable?

47. Explain how even nonrepeated multiple inheritance introduces the need for "this correction" fields in individual vtable entries.

48. Explain how shared multiple inheritance introduces the need for an additional level of indirection when accessing fields of certain parent classes.

49. Explain why true multiple inheritance is harder to implement than interface inheritance, traits, or mix-ins.

10 Object Orientation

10.7.1 The Object Model of Smalltalk

Smalltalk is heavily integrated into its programming environment. In fact, unlike all of the other languages mentioned in this book, a Smalltalk program does not consist of a simple sequence of characters. Rather, Smalltalk programs are meant to be viewed within the *browser* of a Smalltalk implementation, where font changes and screen position can be used to differentiate among various parts of a given program unit. Together with the contemporaneous Interlisp and Pilot/Mesa projects at PARC, the Smalltalk group shares credit for developing the now ubiquitous concepts of bit-mapped screens, windows, menus, and mice.

Smalltalk uses an untyped reference model for all variables. Every variable refers to an object, but the class of the object need not be statically known. As described in Section 10.3.1, every Smalltalk object is an instance of a class descended from a single base class named `Object`. All data are contained in objects. The most trivial of these are simple immutable objects such as `true` (of class `Boolean`) and `3` (of class `Integer`).

EXAMPLE 10.68
Operations as messages in Smalltalk

Operations are all conceptualized as *messages* sent to objects. The expression `3 + 4`, for example, indicates sending a `+` message to the (immutable) object `3`, with a reference to the object `4` as argument. In response to this message, the object `3` creates and returns a reference to the (immutable) object `7`. Similarly, the expression `a + b`, where `a` and `b` are variables, indicates sending a `+` message to the object referred to by `a`, with the reference in `b` as argument. If `a` happens to refer to `3` and `b` refers to `4`, the effect will be the same as it was in the case of the constants. ■

EXAMPLE 10.69
Mixfix messages

As described in Section 6.1, multiargument messages have multiword ("mixfix") names. Each word ends with a colon; each argument follows a word. The expression

```
myBox displayOn: myScreen at: location
```

sends a `displayOn: at:` message to the object referred to by variable `myBox`, with the objects referred to by `myScreen` and `location` as arguments. ■

EXAMPLE 10.70

Selection as an `ifTrue: ifFalse:` message

Even control flow in Smalltalk is conceptualized as messages. Consider the selection construct:

```
n < 0
    ifTrue: [abs <- n negated]
    ifFalse: [abs <- n]
```

This code begins by sending a `< 0` message (a `<` message with `0` as argument) to the object referred to by `n`. In response to this message, the object referred to by `n` will return a reference to one of two immutable objects: `true` or `false`. This reference becomes the value of the `n < 0` expression.

Smalltalk evaluates expressions left-to-right without precedence or associativity. The value of `n < 0` therefore becomes the recipient of an `ifTrue: ifFalse:` message. This message has two arguments, each of which is a *block*. A block in Smalltalk is a fragment of code enclosed in brackets. It is an immutable object, with semantics roughly comparable to those of a lambda expression in Lisp. To execute a block we send it a `value` message.

When sent an `ifTrue: ifFalse:` message, the immutable object `true` sends a `value` message to its first argument (which had better be a block) and then returns the result. The object `false`, on the other hand, in response to the same message, sends a `value` message to its second argument (the block that followed `ifFalse:`). The left arrow (`<-`) in each block is the assignment operator. Assignment is not a message; it is a side effect of evaluation of the right-hand side. As in expression-based languages such as Algol 68, the value of an assignment expression is the value of the right-hand side. The overall value of our selection expression will be the value of one of the blocks, namely a reference to `n` or to its additive inverse, whichever is non-negative. For the sake of convenience, Boolean objects in Smalltalk also implement `ifTrue:`, `ifFalse:`, and `ifFalse: ifTrue:` methods.

EXAMPLE 10.71

Iterating with messages

Iteration is modeled in a similar fashion. For enumeration-controlled loops, class `Integer` implements `timesRepeat:` and `to: by: do:` methods:

```
pow <- 1.
10 timesRepeat:
    [pow <- pow * n]

sum <- 0.
1 to: 100 by: 2 do:
    [:i | sum <- sum + (a at: i)]
```

The first of these code fragments calculates $\mathtt{n}^{10}$. In response to a `timesRepeat:` message, the integer *k* sends a `value` message to the argument (a block) *k* times. The second code fragment sums the odd-indexed elements of the array referred to by `a`. In response to a `to: by: do:` message, the integer *k* behaves as one might expect: it sends a `value:` message to its third argument (a block) $\lfloor (t - k + b)/b \rfloor$ times, where *t* is the first argument and *b* is the second argument. Note the colon at the end of `value:`. The plain `value` message is unary; the `value:` message has an argument; it is understood by blocks that have a (single) formal parameter. In our loop example, the integer `1` sends the messages `value: 1`, `value: 3`, `value: 5`, and so on to the block `[:i | sum <- sum + (a at: i)]`. The `:i |` at the beginning of the block is its formal parameter. The `at:` message is understood by arrays. For iteration with a step size of one, integers also provide a `to: do:` method.

EXAMPLE 10.72
Blocks as closures

Because it is an object, a block can be referred to by a variable:

```
b <- [n <- n + 1].            " b is now a closure"
c <- [:i | n <- n + i].       " so is c"
...
b value.                      " increment n by 1"
c value: 3.                   " increment n by 3"
```

A block with two parameters expects a `value: value:` message. A block with *j* parameters expects a message whose name consists of the word `value:` repeated *j* times. Comments in Smalltalk are double-quoted (strings are single-quoted).

EXAMPLE 10.73
Logical looping with messages

For logically controlled loops, Smalltalk relies on the `whileTrue:` message, understood by blocks:

```
tail <- myList.
[tail next ~~ nil]
    whileTrue: [tail <- tail next]
```

This code sets `tail` to the final element of `myList`. The double-tilde (`~~`) operator means "does not refer to the same object as." The method `next` is assumed to return a reference to the element following its recipient. In response to a `whileTrue:` message, a block sends itself a `value` message. If the result of that message is a reference to `true`, the block sends a `value` message to the argument of the original message and repeats. Blocks also implement a `whileFalse:` method.

EXAMPLE 10.74
Defining control abstractions

The blocks of Smalltalk allow the programmer to construct almost arbitrary control-flow constructs. Because of their simple syntax, Smalltalk blocks are even easier to manipulate than the lambda expressions of Lisp. In effect, a `to: by: do:` message turns iteration "inside out," making the body of the loop a simple message argument that can be executed (by sending it a `value` message) from within the body of the `to: by: do:` method. Smalltalk programmers can define similar methods for other container classes, obtaining all the power of iterators (Section 6.5.3) and much of the power of `call_with_current_continuation` (Section 9.4.3):

```
myTree inorderDo: [:node | whatever ]
```

It is worth noting that the uniform object model of computation in Smalltalk does not necessarily imply a uniform implementation. Just as Clu implementations implement built-in immutable objects as values, despite their reference semantics (Section 6.1.2), a Smalltalk implementation is likely to use the usual machine instructions for computer arithmetic, rather than actually sending messages to integers. In a similar vein, the most common control-flow constructs (`ifTrue: ifFalse:`, `to: by: do:`, `whileTrue:`, etc.) are likely to be recognized by a Smalltalk interpreter, and implemented with special, faster code.

EXAMPLE 10.75

Recursion in Smalltalk

We end this subsection by observing that recursion works at least as well in Smalltalk as it does in other imperative languages. The following is a recursive implementation of Euclid's algorithm:

```
gcd: other                                  "other is a formal parameter"
    (self = other)
        ifTrue:   [↑self].                  "end condition"
    (self < other)
        ifTrue:   [↑self gcd: (other - self)]        "recurse"
        ifFalse:  [↑other gcd: (self - other)]       "recurse"
```

The up-arrow (↑) symbol is comparable to the `return` of C or Algol 68. The keyword `self` is comparable to `this` in C++. We have shown the code in mixed fonts, much as it would appear in a Smalltalk browser. The header of the method is identified by bold face type.

CHECK YOUR UNDERSTANDING

50. Name the three projects at Xerox PARC in the 1970s that pioneered modern GUI-based personal computers.
51. Explain the concept of a *message* in Smalltalk.
52. How does Smalltalk indicate multiple message arguments?
53. What is a *block* in Smalltalk? What mechanism does it resemble in Lisp?
54. Give three examples of how Smalltalk models control flow as message evaluation.
55. Explain how type checking works in Smalltalk.

10 Object Orientation

10.9 Exercises

10.23 Suppose that class D inherits from classes A, B, and C, none of which share any common ancestor. Show how the data members and vtable(s) of D might be laid out in memory. Also show how to convert a reference to a D object into a reference to an A, B, or C object.

10.24 Consider the person_interface and system_user_interface classes described in Example C-10.61. If student is derived from person_interface and system_user_interface, explain what happens in the following method call:

```
student s;
person *p = &s;
...
p->print_stats();
```

You may wish to use a diagram of the representation of a student object to illustrate the method lookups that occur and the views that are computed. You may assume an implementation akin to that of Figure C-10.9, without shared inheritance.

10.25 Given the inheritance tree of Example C-10.62, show a representation for objects of class student_prof. You may want to consult Figures C-10.8, C-10.9, and C-10.10.

10.26 Given the memory layout of Figure C-10.8 and the following declarations:

```
student& sr;
system_user& ur;
```

show the code that must be generated for the assignment

```
ur = sr;
```

(Pitfall: Be sure to consider null pointers.)

10.27 Standard C++ provides a "pointer-to-member" mechanism for classes:

```
class C {
public:
    int a;
    int b;
} c;
int C::*pm = &C::a;
    // pm points to member a of an (arbitrary) C object
...
C* p = &c;
p->*pm = 3;     // assign 3 into c.a
```

Pointers to members are also permitted for subroutine members (methods), including virtual methods. How would you implement pointers to virtual methods in the presence of C++-style multiple inheritance?

10.28 As an alternative to using ⟨method address, `this` correction⟩ pairs in the vtable entries of a language with multiple inheritance, we could leave the entries as simple pointers, but make them point to code that updates `this` in-line, and then jumps to the beginning of the appropriate method. Show the sequence of instructions executed under this scheme. What factors will influence whether it runs faster or slower than the sequence shown in Example C-10.59? Which scheme will use less space? (Remember to count both code and data structure size, and consider which instructions must be replicated at every call site.)

Pursuing the replacement of data structures with executable code even further, consider an implementation in which the vtable itself consists of executable code. Show what this code would look like and, again, discuss the implications for time and space overhead.

10.29 In Eiffel, shared inheritance is the default rather than the exception. Only renamed features are replicated. As a result, it is not possible to tell when looking at a class whether its members will be inherited replicated or shared by derived classes. Describe a uniform mechanism for looking up members inherited from base classes that will work whether they are replicated *or* shared. (Hint: Consider the use of dope vectors for records containing arrays of dynamic shape, as described in Section 8.2.2. For further details, consult the compiler text of Wilhelm and Maurer [WM95, Sec. 5.3].)

10.30 In Figure C-10.10, consider calls to virtual methods declared in `A`, but called through a `B`, `C`, or `D` object view. We could avoid one level of indirection by appending a copy of the `A` part of the vtable to the `D/B` and `C` parts of the vtable (with suitably adjusted `this` corrections). Give calling sequences for this alternative implementation. In the worst case, how much larger may the vtable be for a class with *n* ancestors?

10.31 Consider the Smalltalk implementation of Euclid's algorithm, presented at the end of Section C-10.7.1. Trace the messages involved in evaluating `4 gcd: 6`.

10 Object Orientation

10.10 Explorations

10.39 Figure out how multiple inheritance is implemented in your local C++ compiler. How closely does it follow the strategy of Sections C-10.6.2 and C-10.6.3? What rationale do you see for any differences?

10.40 Learn how multiple inheritance is implemented in Perl and Python (you might begin by reading Section 14.4.4). Describe the differences with respect to Sections C-10.6.2 and C-10.6.3. Discuss the advantages and drawbacks of dynamic typing in object-oriented languages.

11 Functional Languages

11.7 Theoretical Foundations

EXAMPLE 11.77

Functions as mappings

Mathematically, a function is a single-valued mapping: it associates every element in one set (the *domain*) with (at most) one element in another set (the *range*). In conventional notation, we indicate the domain and range by writing

$$\text{sqrt} : \mathcal{R} \longrightarrow \mathcal{R}$$

We can, of course, have functions of more than one variable—that is, functions whose domains are Cartesian products:

$$\text{plus} : [\mathcal{R} \times \mathcal{R}] \longrightarrow \mathcal{R}$$

■

If a function provides a mapping for every element of the domain, the function is said to be *total*. Otherwise, it is said to be *partial*. Our *sqrt* function is partial: it does not provide a mapping for negative numbers. We could change our definition to make the domain of the function the non-negative numbers, but such changes are often inconvenient, or even impossible: inconvenient because we should like all mathematical functions to operate on $\mathcal{R}$; impossible because we may not know which elements of the domain have mappings and which do not. Consider for example the function f that maps every natural number a to the smallest natural number b such that the digits of the decimal representation of a appear b digits to the right of the decimal point in the decimal expansion of π. Clearly $f(59) = 4$, because $\pi = 3.14159\ldots.$ But what about $f(428945028)$, or in general $f(n)$ for arbitrary n? Absent results from number theory, it is not at all clear how to characterize the values at which f is defined. In such a case a partial function is essential.

EXAMPLE 11.78

Functions as sets

It is often useful to characterize functions as sets or, more precisely, as subsets of the Cartesian product of the domain and the range:

$$\text{sqrt} \subset [\mathcal{R} \times \mathcal{R}]$$
$$\text{plus} \subset [\mathcal{R} \times \mathcal{R} \times \mathcal{R}]$$

We can specify *which* subset using traditional set notation:

$$\text{sqrt} \equiv \left\{(x, y) \in \mathcal{R} \times \mathcal{R} \mid y > 0 \wedge x = y^2\right\}$$
$$\text{plus} \equiv \{(x, y, z) \in \mathcal{R} \times \mathcal{R} \times \mathcal{R} \mid z = x + y\}$$

Note that this sort of definition tells us what the value of a function like sqrt is, but it does *not* tell us how to compute it; more on this distinction below. ■

EXAMPLE 11.79

Functions as powerset elements

One of the nice things about the set-based characterization is that it makes it clear that a function is an ordinary mathematical object. We know that a function from A to B is a subset of $A \times B$. This means that it is an *element* of the *powerset* of $A \times B$—the set of all subsets of $A \times B$, denoted $2^{A \times B}$:

$$\text{sqrt} \in 2^{\mathcal{R} \times \mathcal{R}}$$

Similarly,

$$\text{plus} \in 2^{\mathcal{R} \times \mathcal{R} \times \mathcal{R}}$$

Note the overloading of notation here. The powerset 2^A should not be confused with exponentiation, though it is true that for a finite set A the number of elements in the powerset of A is 2^n, where $n = |A|$, the cardinality of A. ■

EXAMPLE 11.80

Function spaces

Because functions are single-valued, we know that they constitute only *some* of the elements of $2^{A \times B}$. Specifically, they constitute all and only those sets of pairs in which the first component of each pair is unique. We call the set of such sets the *function space* of A into B, denoted $A \rightarrow B$. Note that $(A \rightarrow B) \subset 2^{A \times B}$. In our examples:

$$\text{sqrt} \in [\mathcal{R} \rightarrow \mathcal{R}]$$
$$\text{plus} \in [(\mathcal{R} \times \mathcal{R}) \rightarrow \mathcal{R}]$$

EXAMPLE 11.81

Higher-order functions as sets

Now that functions are elements of sets, we can easily build higher-order functions:

$$\text{compose} \equiv \{(f, g, h) \mid \forall x \in \mathcal{R},\ h(x) = f(g(x))\}$$

What are the domain and range of compose? We know that f, g, and h are elements of $\mathcal{R} \rightarrow \mathcal{R}$. Thus

$$\text{compose} \in [(\mathcal{R} \rightarrow \mathcal{R}) \times (\mathcal{R} \rightarrow \mathcal{R})] \rightarrow (\mathcal{R} \rightarrow \mathcal{R})$$

Note the similarity to the notation employed by the ML type system (Section 7.4). ■

EXAMPLE 11.82

Curried functions as sets

Using the notion of "currying" from Section 11.6, we note that there is an alternative characterization for functions like plus. Rather than a function from pairs of reals to reals, we can capture it as a function from reals to functions from reals to reals:

$$\text{curried\_plus} \in \mathcal{R} \rightarrow (\mathcal{R} \rightarrow \mathcal{R})$$ ■

We shall have more to say about currying in Section C-11.7.3.

11.7.1 Lambda Calculus

As we suggested in the main text, one of the limitations of the function-as-set notation is that it is *nonconstructive*: it doesn't tell us how to *compute* the value of a function at a given point (i.e., on a given input). Church designed the lambda calculus to address this limitation. In its pure form, lambda calculus represents *everything* as a function. The natural numbers, for example, can be represented by a distinguished zero function (commonly the identity function) and a successor function. (One common formulation uses a select_second function that takes two arguments and returns the second of them. The successor function is then defined in such a way that the number n ends up being represented by a function that, when applied to select_second n times, returns the identity function [Mic89, Sec. 3.5]; [Sta95, Sec. 7.6]; see Exercise C-11.23.) While of theoretical importance, this formulation of arithmetic is highly cumbersome. We will therefore take ordinary arithmetic as a given in the remainder of this subsection. (And of course all practical functional programming languages provide built-in support for both integer and floating-point arithmetic.)

A lambda expression can be defined recursively as (1) a *name*; (2) a lambda *abstraction* consisting of the letter λ, a name, a dot, and a lambda expression; (3) a function *application* consisting of two adjacent lambda expressions; or (4) a parenthesized lambda expression. To accommodate arithmetic, we will extend this definition to allow numeric literals.

EXAMPLE 11.83
Juxtaposition as function application

When two expressions appear adjacent to one another, the first is interpreted as a function to be applied to the second:

sqrt n

Most authors assume that application associates left-to-right (so $f\ A\ B$ is interpreted as $(f\ A)\ B$, rather than $f\ (A\ B)$), and that application has higher precedence than abstraction (so $\lambda x.A\ B$ is interpreted as $\lambda x.(A\ B)$, rather than $(\lambda x.A)\ B$). ML adopts these rules. ■

EXAMPLE 11.84
Lambda calculus syntax

Parentheses are used as necessary to override default groupings. Specifically, if we distinguish between lambda expressions that are used as functions and those that are used as arguments, then the following unambiguous CFG can be used to generate lambda expressions with a minimal number of parentheses:

$$\begin{aligned}
\mathit{expr} &\longrightarrow \texttt{name} \mid \mathit{number} \mid \lambda\ \texttt{name}\ \texttt{.}\ \mathit{expr} \mid \ \mathit{func}\ \mathit{arg} \\
\mathit{func} &\longrightarrow \texttt{name} \mid \texttt{(}\ \lambda\ \texttt{name}\ \texttt{.}\ \mathit{expr}\ \texttt{)} \mid \ \mathit{func}\ \mathit{arg} \\
\mathit{arg} &\longrightarrow \texttt{name} \mid \mathit{number} \mid \texttt{(}\ \lambda\ \texttt{name}\ \texttt{.}\ \mathit{expr}\ \texttt{)} \mid \texttt{(}\ \mathit{func}\ \mathit{arg}\ \texttt{)}
\end{aligned}$$

In words: we use parentheses to surround an abstraction that is used as either a function or an argument, and around an application that is used as an argument. ■

EXAMPLE 11.85
Binding parameters with λ

The letter λ introduces the lambda calculus equivalent of a formal parameter. The following lambda expression denotes a function that returns the square of its argument:

$$\lambda x.\mathsf{times}\ x\ x$$

The name (variable) introduced by a λ is said to be *bound* within the expression following the dot. In programming language terms, this expression is the variable's scope. A variable that is not bound is said to be *free*.

EXAMPLE 11.86
Free variables

As in a lexically scoped programming language, a free variable needs to be defined in some surrounding scope. Consider, for example, the expression $\lambda x.\lambda y.\mathsf{times}\ x\ y$. In the inner expression ($\lambda y.\mathsf{times}\ x\ y$), y is bound but x is free. There are no restrictions on the use of a bound variable: it can play the role of a function, an argument, or both. Higher-order functions are therefore completely natural.

EXAMPLE 11.87
Naming functions for future reference

If we wish to refer to them later, we can give expressions names:

$$\begin{aligned}
\mathsf{square} &\equiv \lambda x.\mathsf{times}\ x\ x\\
\mathsf{identity} &\equiv \lambda x.x\\
\mathsf{const7} &\equiv \lambda x.7\\
\mathsf{hypot} &\equiv \lambda x.\lambda y.\mathsf{sqrt}\ (\mathsf{plus}\ (\mathsf{square}\ x)\ (\mathsf{square}\ y))
\end{aligned}$$

Here $\equiv$ is a metasymbol meaning, roughly, "is an abbreviation for."

EXAMPLE 11.88
Evaluation rules

To compute with the lambda calculus, we need rules to evaluate expressions. It turns out that three rules suffice:

beta reduction: For any lambda abstraction $\lambda x.E$ and any expression M, we say

$$(\lambda x.E)\ M \rightarrow_{\beta} E[M\backslash x]$$

where $E[M\backslash x]$ denotes the expression E with all free occurrences of x replaced by M. Beta reduction is not permitted if any free variables in M would become bound in $E[M\backslash x]$.

alpha conversion: For any lambda abstraction $\lambda x.E$ and any variable y that has no free occurrences in E, we say

$$\lambda x.E \rightarrow_{\alpha} \lambda y.E[y\backslash x]$$

eta reduction: A rule to eliminate "surplus" lambda abstractions. For any lambda abstraction $\lambda x.E$, where E is of the form $F\,x$, and x has no free occurrences in F, we say

$$\lambda x.F\,x \rightarrow_{\eta} F$$

EXAMPLE 11.89
Delta reduction for arithmetic

To accommodate arithmetic we will also allow an expression of the form $\mathsf{op}\ x\ y$, where x and y are numeric literals and op is one of a small set of standard functions, to be replaced by its arithmetic value. This replacement is called *delta reduction*. In our examples we will need only the functions plus, minus, and times:

$$\begin{array}{lll}
 & (\lambda \underline{f}.\lambda g.\lambda h.f\,g(h\,h))\underline{(\lambda x.\lambda y.x)}h(\lambda x.x\,x) & \\
\rightarrow_\beta & (\lambda g.\lambda \underline{h}.(\lambda x.\lambda y.x)g(\underline{h\,h}))h(\lambda x.x\,x) & (1)\\
\rightarrow_\alpha & (\lambda \underline{g}.\lambda k.(\lambda x.\lambda y.x)g(k\,k))\underline{h}(\lambda x.x\,x) & (2)\\
\rightarrow_\beta & (\lambda \underline{k}.(\lambda x.\lambda y.x)h(k\,k))\underline{(\lambda x.x\,x)} & (3)\\
\rightarrow_\beta & (\lambda \underline{x}.\lambda y.x)\underline{h}((\lambda x.x\,x)\ (\lambda x.x\,x)) & (4)\\
\rightarrow_\beta & (\lambda \underline{y}.h)\underline{((\lambda x.x\,x)\ (\lambda x.x\,x))} & (5)\\
\rightarrow_\beta & h & (6)
\end{array}$$

Figure 11.5 **Reduction of a lambda expression.** The top line consists of a function applied to three arguments. The first argument (underlined) is the "select first" function, which takes two arguments and returns the first. The second argument is the symbol h, which must be either a constant or a variable bound in some enclosing scope (not shown). The third argument is an "apply to self" function that takes one argument and applies it to itself. The particular series of reductions shown occurs in normal order. It terminates with a simplest (normal) form of simply h.

$$\begin{array}{rcl}
\text{plus}\ 2\ 3 & \rightarrow_\delta & 5\\
\text{minus}\ 5\ 2 & \rightarrow_\delta & 3\\
\text{times}\ 2\ 3 & \rightarrow_\delta & 6
\end{array}$$

EXAMPLE 11.90
Eta reduction

Beta reduction resembles the use of call by name parameters (Section 9.3.1). Unlike Algol 60, however, the lambda calculus provides no way for an argument to carry its referencing environment with it; hence the requirement that an argument not move a variable into a scope in which its name has a different meaning. Alpha conversion serves to change names to make beta reduction possible. Eta reduction is comparatively less important. If square is defined as above, eta reduction allows us to say that

$$\lambda x.\text{square}\ x \rightarrow_\eta \text{square}$$

In English, square is a function that squares its argument; $\lambda x.$square x is a function of x that squares x. The latter reminds us explicitly that it's a function (i.e., that it takes an argument), but the former is a little less messy looking.

EXAMPLE 11.91
Reduction to simplest form

Through repeated application of beta reduction and alpha conversion (and possibly eta reduction), we can attempt to reduce a lambda expression to its simplest possible form—a form in which no further beta reductions are possible. An example can be found in Figure C-11.5. In line (2) of this derivation we have to employ an alpha conversion because the argument that we need to substitute for g contains a free variable (h) that is bound within g's scope. If we were to make the substitution of line (3) without first having renamed the bound h (as k), then the free h would have been *captured*, erroneously changing the meaning of the expression.

In line (5) of the derivation, we had a choice as to which subexpression to reduce. At that point the expression as a whole consisted of a function application in which the argument was itself a function application. We chose to substitute the main argument $((\lambda x.x\,x)\;(\lambda x.x\,x))$, unevaluated, into the body of the main lambda abstraction. This choice is known as *normal-order* reduction, and corresponds to normal-order evaluation of arguments in programming languages, as discussed in Sections 6.6.2 and 11.5. In general, whenever more than one beta reduction could be made, normal order chooses the one whose λ is left-most in the overall expression. This strategy substitutes arguments into functions before reducing them. The principal alternative, *applicative-order* reduction, reduces both the function part and the argument part of every function application to the simplest possible form before substituting the latter into the former. ■

Church and Rosser showed in 1936 that simplest forms are unique: any series of reductions that terminates in a nonreducible expression will produce the same result. Not all reductions terminate, however. In particular, there are expressions for which no series of reductions will terminate, and there are others in which normal-order reduction will terminate but applicative-order reduction will not.

EXAMPLE 11.92

Nonterminating applicative-order reduction

The example expression of Figure C-11.5 leads to an infinite "computation" under applicative-order reduction. To see this, consider the expression at line (5). This line consists of the constant function $(\lambda y.h)$ applied to the argument $(\lambda x.x\,x)\;(\lambda x.x\,x)$. If we attempt to evaluate the argument before substituting it into the function, we run through the following steps:

$$\begin{array}{rl}
 & (\underline{\lambda x}.x\,x)\;\underline{(\lambda x.x\,x)} \\
\rightarrow_\beta & (\underline{\lambda x}.x\,x)\;\underline{(\lambda x.x\,x)} \\
\rightarrow_\beta & (\underline{\lambda x}.x\,x)\;\underline{(\lambda x.x\,x)} \\
\rightarrow_\beta & (\underline{\lambda x}.x\,x)\;\underline{(\lambda x.x\,x)} \\
\dots &
\end{array}$$

■

In addition to showing the uniqueness of simplest (normal) forms, Church and Rosser showed that if any evaluation order will terminate, normal order will. This pair of results is known as the *Church-Rosser theorem.*

11.7.2 Control Flow

We noted at the beginning of the previous subsection that arithmetic can be modeled in the lambda calculus using a distinguished zero function (commonly the identity) and a successor function. What about control-flow constructs—selection and recursion in particular?

EXAMPLE 11.93

Booleans and conditionals

The select_first function, $\lambda x.\lambda y.x$, is commonly used to represent the Boolean value true. The select_second function, $\lambda x.\lambda y.y$, is commonly used to represent the Boolean value false. Let us denote these by T and F. The nice thing about these definitions is that they allow us to define an if function very easily:

$$\mathsf{if} \equiv \lambda c.\lambda t.\lambda e.c\ t\ e$$

Consider:

$$\begin{aligned}
\mathsf{if}\ T\ 3\ 4 &\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)\ (\lambda x.\lambda y.x)\ 3\ 4 \\
&\rightarrow_\beta^* (\lambda x.\lambda y.x)\ 3\ 4 \\
&\rightarrow_\beta^* 3
\end{aligned}$$

$$\begin{aligned}
\mathsf{if}\ F\ 3\ 4 &\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)\ (\lambda x.\lambda y.y)\ 3\ 4 \\
&\rightarrow_\beta^* (\lambda x.\lambda y.y)\ 3\ 4 \\
&\rightarrow_\beta^* 4
\end{aligned}$$

■

Functions like equal and greater_than can be defined to take numeric values as arguments, returning T or F.

EXAMPLE 11.94

Beta abstraction for recursion

Recursion is a little tricky. An equation like

$$\begin{aligned}
\mathsf{gcd} \equiv\ & \lambda a.\lambda b.(\mathsf{if}\ (\mathsf{equal}\ a\ b)\ a \\
& \quad (\mathsf{if}\ (\mathsf{greater\_than}\ a\ b)\ (\mathsf{gcd}\ (\mathsf{minus}\ a\ b)\ b)\ (\mathsf{gcd}\ (\mathsf{minus}\ b\ a)\ a)))
\end{aligned}$$

is not really a definition at all, because gcd appears on both sides. Our previous definitions (T, F, if) were simply shorthand: we could substitute them out to obtain a pure lambda expression. If we try that with gcd, the "definition" just gets bigger, with new occurrences of the gcd name. To obtain a real definition, we first rewrite our equation using *beta abstraction* (the opposite of beta reduction):

$$\begin{aligned}
\mathsf{gcd} \equiv\ & (\lambda g.\lambda a.\lambda b.(\mathsf{if}\ (\mathsf{equal}\ a\ b)\ a \\
& \quad (\mathsf{if}\ (\mathsf{greater\_than}\ a\ b)\ (g(\mathsf{minus}\ a\ b)\ b)\ (g(\mathsf{minus}\ b\ a)\ a))))\ \mathsf{gcd}
\end{aligned}$$

Now our equation has the form

$$\mathsf{gcd} \equiv f\ \mathsf{gcd}$$

where f is the perfectly well-defined (nonrecursive) lambda expression

$$\begin{aligned}
& \lambda g.\lambda a.\lambda b.(\mathsf{if}\ (\mathsf{equal}\ a\ b)\ a \\
& \quad (\mathsf{if}\ (\mathsf{greater\_than}\ a\ b)\ (g\ (\mathsf{minus}\ a\ b)\ b)\ (g\ (\mathsf{minus}\ b\ a)\ a)))
\end{aligned}$$

Clearly gcd is a fixed point of f. ■

EXAMPLE 11.95

The fixed-point combinator **Y**

As it turns out, for any function f given by a lambda expression, we can find the least (simplest) fixed point of f, if there is a fixed point, by applying the *fixed-point combinator*

$$\lambda h.(\lambda x.h(xx))\ (\lambda x.h(xx))$$

commonly denoted **Y**. **Y** has the property that for any lambda expression f, if the normal-order evaluation of $\mathbf{Y}f$ terminates, then $f(\mathbf{Y}f)$ and $\mathbf{Y}f$ will reduce to the same simplest form (see Exercise C-11.21). In the case of our gcd function, we have

$$\begin{aligned}
\text{gcd} \equiv\ & (\lambda h.(\lambda x.h(x\,x))\,(\lambda x.h(x\,x))) \\
& (\lambda g.\lambda a.\lambda b.(\text{if }(\text{equal } a\, b)\ a \\
& \quad (\text{if }(\text{greater\_than } a\, b)\ (g(\text{minus } a\, b)\ b)\ (g(\text{minus } b\, a)\ a))))
\end{aligned}$$

Figure C-11.6 traces the evaluation of gcd 4 2. Given the existence of the **Y** combinator, most authors permit recursive "definitions" of functions, for convenience.

11.7.3 Structures

EXAMPLE 11.96

Lambda calculus list operators

Just as we can use functions to build numbers and truth values, we can also use them to encapsulate values in structures. Using Scheme terminology for the sake of clarity, we can define simple list-processing functions as follows:

$$\begin{aligned}
\text{cons} &\equiv \lambda a.\lambda d.\lambda x.x\ a\ d \\
\text{car} &\equiv \lambda l.l\ \text{select\_first} \\
\text{cdr} &\equiv \lambda l.l\ \text{select\_second} \\
\text{nil} &\equiv \lambda x.T \\
\text{null?} &\equiv \lambda l.l(\lambda x.\lambda y.F)
\end{aligned}$$

where select_first and select_second are the functions $\lambda x.\lambda y.x$ and $\lambda x.\lambda y.y$, respectively—functions we also use to represent true and false.

EXAMPLE 11.97

List operator identities

Using these definitions we can see that

$$\begin{aligned}
\text{car}(\text{cons } A\, B) &\equiv (\lambda l.l\ \text{select\_first})\ (\text{cons } A\, B) \\
&\to_\beta (\text{cons } A\, B)\ \text{select\_first} \\
&\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\, B)\ \text{select\_first} \\
&\to_\beta^* (\lambda x.x\ A\, B)\ \text{select\_first} \\
&\to_\beta \text{select\_first}\ A\, B \\
&\equiv (\lambda x.\lambda y.x)\ A\, B \\
&\to_\beta^* A
\end{aligned}$$

$$\begin{aligned}
\text{cdr}(\text{cons } A\, B) &\equiv (\lambda l.l\ \text{select\_second})\ (\text{cons } A\, B) \\
&\to_\beta (\text{cons } A\, B)\ \text{select\_second} \\
&\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\, B)\ \text{select\_second} \\
&\to_\beta^* (\lambda x.x\ A\, B)\ \text{select\_second} \\
&\to_\beta \text{select\_second}\ A\, B \\
&\equiv (\lambda x.\lambda y.y)\ A\, B \\
&\to_\beta^* B
\end{aligned}$$

$$
\begin{aligned}
\text{gcd}\ 2\ 4 \quad &\equiv \quad \mathbf{Y} f\ 2\ 4 \\
&\equiv \quad ((\lambda h.(\lambda x.h(x\,x))\ (\lambda x.h(x\,x)))f)\ 2\ 4 \\
&\rightarrow_\beta \quad ((\lambda x.f(x\,x))\ (\lambda x.f(x\,x)))\ 2\ 4 \\
&\equiv \quad (k\,k)\ 2\ 4,\ \ \text{where}\ k \equiv \lambda x.f(x\,x) \\
&\rightarrow_\beta \quad (f(k\,k))\ 2\ 4 \\
&\equiv \quad ((\lambda g.\lambda a.\lambda b.(\text{if}\ (=\ a\ b)\ a\ (\text{if}\ (>a\ b)\ (g(-\ a\ b)\ b)\ (g(-\ b\ a)\ a))))\ (k\,k))\ 2\ 4 \\
&\rightarrow_\beta \quad (\lambda a.\lambda b.(\text{if}\ (=\ a\ b)\ a\ (\text{if}\ (>a\ b)\ ((k\,k)(-\ a\ b)\ b)\ ((k\,k)(-\ b\ a)\ a))))\ 2\ 4 \\
&\rightarrow_\beta^* \quad \text{if}\ (=\ 2\ 4)\ 2\ (\text{if}\ (>2\ 4)\ ((k\,k)\ (-\ 2\ 4)\ 4)\ ((k\,k)\ (-\ 4\ 2)\ 2)) \\
&\equiv \quad (\lambda c.\lambda t.\lambda e.c\ t\ e)\ (=\ 2\ 4)\ 2\ (\text{if}\ (>2\ 4)\ ((k\,k)\ (-\ 2\ 4)\ 4)\ ((k\,k)\ (-\ 4\ 2)\ 2)) \\
&\rightarrow_\beta^* \quad (=\ 2\ 4)\ 2\ (\text{if}\ (>2\ 4)\ ((k\,k)\ (-\ 2\ 4)\ 4)\ ((k\,k)\ (-\ 4\ 2)\ 2)) \\
&\rightarrow_\delta \quad F\ 2\ (\text{if}\ (>2\ 4)\ ((k\,k)\ (-\ 2\ 4)\ 4)\ ((k\,k)\ (-\ 4\ 2)\ 2)) \\
&\equiv \quad (\lambda x.\lambda y.y)\ 2\ (\text{if}\ (>2\ 4)\ ((k\,k)\ (-\ 2\ 4)\ 4)\ ((k\,k)\ (-\ 4\ 2)\ 2)) \\
&\rightarrow_\beta^* \quad \text{if}\ (>2\ 4)\ ((k\,k)\ (-\ 2\ 4)\ 4)\ ((k\,k)\ (-\ 4\ 2)\ 2) \\
&\rightarrow \quad \ldots \\
&\rightarrow \quad (k\,k)\ (-\ 4\ 2)\ 2 \\
&\equiv \quad ((\lambda x.f(x\,x))k)\ (-\ 4\ 2)\ 2 \\
&\rightarrow_\beta \quad (f(k\,k))\ (-\ 4\ 2)\ 2 \\
&\equiv \quad ((\lambda g.\lambda a.\lambda b.(\text{if}\ (=\ a\ b)\ a\ (\text{if}\ (>a\ b)\ (g(-\ a\ b)\ b)\ (g(-\ b\ a)\ a))))\ (k\,k))\ (-\ 4\ 2)\ 2 \\
&\rightarrow_\beta \quad (\lambda a.\lambda b.(\text{if}\ (=\ a\ b)\ a\ (\text{if}\ (>a\ b)\ ((k\,k)(-\ a\ b)\ b)\ ((k\,k)(-\ b\ a)\ a))))\ (-\ 4\ 2)\ 2 \\
&\rightarrow_\beta^* \quad \text{if}\ (=\ (-\ 4\ 2)\ 2)\ (-\ 4\ 2)\ (\text{if}\ (>(-\ 4\ 2)\ 2)\ ((k\,k)\ (-\ (-\ 4\ 2)\ 2)\ 2)\ ((k\,k)\ (-\ 2\ (-\ 4\ 2))\ (-\ 4\ 2))) \\
&\equiv \quad (\lambda c.\lambda t.\lambda e.c\ t\ e) \\
&\qquad (=\ (-\ 4\ 2)\ 2)\ (-\ 4\ 2)\ (\text{if}\ (>(-\ 4\ 2)\ 2)\ ((k\,k)\ (-\ (-\ 4\ 2)\ 2)\ 2)\ ((k\,k)\ (-\ 2\ (-\ 4\ 2))\ (-\ 4\ 2))) \\
&\rightarrow_\beta^* \quad (=\ (-\ 4\ 2)\ 2)\ (-\ 4\ 2)\ (\text{if}\ (>(-\ 4\ 2)\ 2)\ ((k\,k)\ (-\ (-\ 4\ 2)\ 2)\ 2)\ ((k\,k)\ (-\ 2\ (-\ 4\ 2))\ (-\ 4\ 2))) \\
&\rightarrow_\delta \quad (=\ 2\ 2)\ (-\ 4\ 2)\ (\text{if}\ (>(-\ 4\ 2)\ 2)\ ((k\,k)\ (-\ (-\ 4\ 2)\ 2)\ 2)\ ((k\,k)\ (-\ 2\ (-\ 4\ 2))\ (-\ 4\ 2))) \\
&\rightarrow_\delta \quad T\ (-\ 4\ 2)\ (\text{if}\ (>(-\ 4\ 2)\ 2)\ ((k\,k)\ (-\ (-\ 4\ 2)\ 2)\ 2)\ ((k\,k)\ (-\ 2\ (-\ 4\ 2))\ (-\ 4\ 2))) \\
&\equiv \quad (\lambda x.\lambda y.x)\ (-\ 4\ 2)\ (\text{if}\ (>(-\ 4\ 2)\ 2)\ ((k\,k)\ (-\ (-\ 4\ 2)\ 2)\ 2)\ ((k\,k)\ (-\ 2\ (-\ 4\ 2))\ (-\ 4\ 2))) \\
&\rightarrow_\beta^* \quad (-\ 4\ 2) \\
&\rightarrow_\delta \quad 2
\end{aligned}
$$

Figure 11.6 Evaluation of a recursive lambda expression. As explained in the body of the text, gcd is defined to be the fixed-point combinator **Y** applied to a beta abstraction f of the standard recursive definition for greatest common divisor. Specifically, **Y** is $\lambda h.(\lambda x.h(x\,x))\ (\lambda x.h(x\,x))$ and f is $\lambda g.\lambda a.\lambda b.(\text{if}\ (=a\ b)\ a\ (\text{if}\ (>a\ b)\ (g(-\ a\ b)\ b)\ (g(-\ b\ a)\ a)))$. For brevity we have used =, >, and − in place of equal, greater_than, and minus. We have performed the evaluation in normal order.

$$\begin{aligned}
\textsf{null? nil} &\equiv (\lambda l.l\ (\lambda x.\lambda y.\textsf{select\_second}))\ \textsf{nil}\\
&\to_\beta \textsf{nil}\ (\lambda x.\lambda y.\textsf{select\_second})\\
&\equiv (\lambda x.\textsf{select\_first})\ (\lambda x.\lambda y.\textsf{select\_second})\\
&\to_\beta \textsf{select\_first}\\
&\equiv T
\end{aligned}$$

$$\begin{aligned}
\textsf{null? (cons}\ A\ B\textsf{)} &\equiv (\lambda l.l\ (\lambda x.\lambda y.\textsf{select\_second}))\ (\textsf{cons}\ A\ B)\\
&\to_\beta (\textsf{cons}\ A\ B)\ (\lambda x.\lambda y.\textsf{select\_second})\\
&\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ (\lambda x.\lambda y.\textsf{select\_second})\\
&\to_\beta^* (\lambda x.x\ A\ B)\ (\lambda x.\lambda y.\textsf{select\_second})\\
&\to_\beta (\lambda x.\lambda y.\textsf{select\_second})\ A\ B\\
&\to_\beta^* \textsf{select\_second}\\
&\equiv F
\end{aligned}$$

EXAMPLE 11.98

Nesting of lambda expressions

Because every lambda abstraction has a single argument, lambda expressions are naturally curried. We generally obtain the effect of a multiargument function by nesting lambda abstractions:

$$\textsf{compose} \equiv \lambda f.\lambda g.\lambda x.f\ (g\ x)$$

which groups as

$$\lambda f.(\lambda g.(\lambda x.(f\ (g\ x))))$$

We commonly think of compose as a function that takes two functions as arguments and returns a third function as its result. We could just as easily, however, think of compose as a function of three arguments: the f, g, and x above. The official story, or course, is that compose is a function of one argument that evaluates to a function of one argument that in turn evaluates to a function of one argument.

EXAMPLE 11.99

Paired arguments and currying

If desired, we can use our structure-building functions to define a noncurried version of compose whose (single) argument is a pair:

$$\textsf{paired\_compose} \equiv \lambda p.\lambda x.(\textsf{car}\ p)\ ((\textsf{cdr}\ p)\ x)$$

If we consider the pairing of arguments as a general technique, we can write a curry function that reproduces the single-argument version, just as we did in Scheme in Section 11.6:

$$\textsf{curry} \equiv \lambda f.\lambda a.\lambda b.f(\textsf{cons}\ a\ b)$$

CHECK YOUR UNDERSTANDING

29. What is the difference between *partial* and *total* functions? Why is the difference important?
30. What is meant by the *function space* $A \to B$?
31. Define *beta reduction*, *alpha conversion*, *eta reduction*, and *delta reduction*.
32. How does beta reduction in lambda calculus differ from lazy evaluation of arguments in a nonstrict programming language like Haskell?
33. Explain how lambda expressions can be used to represent Boolean values and control flow.
34. What is *beta abstraction*?
35. What is the **Y** combinator? What useful property does it possess?
36. Explain how lambda expressions can be used to represent structured values such as lists.
37. State the *Church-Rosser theorem*.

11 Functional Languages

11.10 Exercises

11.20 In Figure C-11.6 we evaluated our expression in normal order. Did we really have any choice? What would happen if we tried to use applicative order?

11.21 Prove that for any lambda expression f, if the normal-order evaluation of $\mathbf{Y}f$ terminates, where $\mathbf{Y}$ is the fixed-point combinator $\lambda h.(\lambda x.h(x\,x))\,(\lambda x.h(x\,x))$, then $f(\mathbf{Y}f)$ and $\mathbf{Y}f$ will reduce to the same simplest form.

11.22 Given the definition of structures (lists) in Section C-11.7.3, what happens if we apply car or cdr to nil? How might you introduce the notion of "type error" into lambda calculus?

11.23 Let

$$\mathsf{zero} \equiv \lambda x.x$$
$$\mathsf{succ} \equiv \lambda n.(\lambda s.(s\ \mathsf{select\_second})\ n)$$

where $\mathsf{select\_second} \equiv \lambda x.\lambda y.y$. Now let

$$\mathsf{one} \equiv \mathsf{succ\ zero}$$
$$\mathsf{two} \equiv \mathsf{succ\ one}$$

Show that

$$\mathsf{one\ select\_second} = \mathsf{zero}$$
$$\mathsf{two\ select\_second\ select\_second} = \mathsf{zero}$$

In general, show that

$$\mathsf{succ}^{n}\ \mathsf{zero\ select\_second}^{n} = \mathsf{zero}$$

Use this result to define a predecessor function pred. You may ignore the issue of the predecessor of zero.

Note that our definitions of T and F allow us to check whether a number is equal to zero:

$$\text{iszero} \equiv \lambda n.(n\ \text{select\_first})$$

Using succ, pred, iszero, and if, show how to define plus and times recursively. These definitions could of course be made nonrecursive by means of beta abstraction and **Y**.

11 Functional Languages

11.11 Explorations

11.30 Learn about the *typed lambda calculus*. What properties does it have that standard lambda calculus does not? What restrictions does it place on permissible expressions? Possible places to start include Cardelli and Wegner's classic survey [CW85] or the newer text by Pierce [Pie02].

11.31 Learn more about *fixed points*. We mentioned these when presenting the **Y** combinator in Section C-11.7.2. They also arise in the denotational definition of loop constructs, in metacircular interpreters [AS96, Sec. 4.1]), and in the *data flow analysis* used by optimizing compilers (Section C-17.4.2). What do these subjects have in common? Are there important differences as well?

11.32 Explore the connection between lexical scoping in Scheme or OCaml and the notion of free and bound variables in lambda calculus. How closely are these related? Why does lambda calculus require alpha conversion but Scheme and OCaml do not? Is there any analogy in lambda calculus to the dynamic scoping of early dialects of Lisp?

12 Logic Languages

12.3 Theoretical Foundations

In mathematical logic, a *predicate* is a function that maps constants (atoms) or variables to the values true and false. *Predicate calculus* provides a notation and inference rules for constructing and reasoning about *propositions* (*statements*) composed of predicate applications, *operators*, and the *quantifiers* $\forall$ and $\exists$.[1] Operators include and ($\wedge$), or ($\vee$), not ($\neg$), implication ($\rightarrow$), and equivalence ($\leftrightarrow$). Quantifiers are used to introduce bound variables in an appended proposition, much as λ introduces variables in the lambda calculus. The *universal* quantifier, $\forall$, indicates that the proposition is true for all values of the variable. The *existential* quantifier, $\exists$, indicates that the proposition is true for at least one value of the variable. Here are a few examples:

EXAMPLE 12.39

Propositions

$$\forall C[\text{rainy}(C) \wedge \text{cold}(C) \rightarrow \text{snowy}(C)]$$

(For all cities C, if C is rainy and C is cold, then C is snowy.)

$$\forall A, \forall B[(\exists C[\text{takes}(A, C) \wedge \text{takes}(B, C)]) \rightarrow \text{classmates}(A, B)]$$

(For all students A and B, if there exists a class C such that A takes C and B takes C, then A and B are classmates.)

$$\forall N[(N > 2) \rightarrow \neg(\exists A, \exists B, \exists C[A^N + B^N = C^N])]$$

(Fermat's last theorem.)

EXAMPLE 12.40

Different ways to say things

One of the interesting characteristics of predicate calculus is that there are many ways to say the same thing. For example,

[1] Strictly speaking, what we are describing here is the *first-order* predicate calculus. There exist higher-order calculi in which predicates can be applied to predicates, not just to atoms and variables. Prolog allows the user to construct higher-order predicates using `call`; the formalization of such predicates is beyond the scope of our coverage here.

$$\begin{aligned}(P_1 \rightarrow P_2) &\equiv (\neg P_1 \vee P_2)\\ (\neg\exists X[P(X)]) &\equiv (\forall X[\neg P(X)])\\ \neg(P_1 \wedge P_2) &\equiv (\neg P_1 \vee \neg P_2)\end{aligned}$$

This flexibility of expression tends to be handy for human beings, but it can be a nuisance for automatic theorem proving. Propositions are much easier to manipulate algorithmically if they are placed in some sort of *normal form*. One popular candidate is known as *clausal form*. We consider this form in the following section. ■

12.3.1 Clausal Form

As it turns out, clausal form is very closely related to the structure of Prolog programs: once we have a proposition in clausal form, it will be relatively easy to translate it into Prolog. We should note at the outset, however, that the translation is not perfect: there are aspects of predicate calculus that Prolog cannot capture, and there are aspects of Prolog (e.g., its imperative and database-manipulating features) that have no analogues in predicate calculus.

Clocksin and Mellish [CM03, Chap. 10] describe a five-step procedure (based heavily on an article by Martin Davis [Dav63]) to translate an arbitrary first-order predicate proposition into clausal form. We trace that procedure here.

EXAMPLE 12.41
Conversion to clausal form

In the first step, we eliminate implication and equivalence operators. As a concrete example, the proposition

$$\forall A[\neg\mathsf{student}(A) \rightarrow (\neg\mathsf{dorm\_resident}(A) \wedge \neg\exists B[\mathsf{takes}(A, B) \wedge \mathsf{class}(B)])]$$

would become

$$\forall A[\mathsf{student}(A) \vee (\neg\mathsf{dorm\_resident}(A) \wedge \neg\exists B[\mathsf{takes}(A, B) \wedge \mathsf{class}(B)])]$$

In the second step, we move negation inward so that the only negated items are individual terms (predicates applied to arguments):

$$\begin{aligned}&\forall A[\mathsf{student}(A) \vee (\neg\mathsf{dorm\_resident}(A) \wedge \forall B[\neg(\mathsf{takes}(A, B) \wedge \mathsf{class}(B))])]\\ \equiv\ &\forall A[\mathsf{student}(A) \vee (\neg\mathsf{dorm\_resident}(A) \wedge \forall B[\neg\mathsf{takes}(A, B) \vee \neg\mathsf{class}(B)])]\end{aligned}$$

In the third step, we use a technique known as Skolemization (due to logician Thoralf Skolem) to eliminate existential quantifiers. We will consider this technique further in Section C-12.3.3. Our example has no existential quantifiers at this stage, so we proceed.

In the fourth step, we move all universal quantifiers to the outside of the proposition (in the absence of naming conflicts, this does not change the proposition's meaning). We then adopt the convention that all variables are universally quantified, and drop the explicit quantifiers:

$$\text{student}(A) \vee (\neg\text{dorm\_resident}(A) \wedge (\neg\text{takes}(A, B) \vee \neg\text{class}(B)))$$

Finally, in the fifth step, we use the distributive, associative, and commutative rules of Boolean algebra to convert the proposition to *conjunctive normal form*, in which the operators $\wedge$ and $\vee$ are nested no more than two levels deep, with $\wedge$ on the outside and $\vee$ on the inside:

$$(\text{student}(A) \vee \neg\text{dorm\_resident}(A)) \wedge (\text{student}(A) \vee \neg\text{takes}(A, B) \vee \neg\text{class}(B))$$

Our proposition is now in clausal form. Specifically, it is in conjunctive normal form, with negation only of individual terms, with no existential quantifiers, and with implied universal quantifiers for all variables (i.e., for all names that are neither constants nor predicates). The clauses are the items at the outer level—the things that are and-ed together.

EXAMPLE 12.42
Conversion to Prolog

To translate the proposition to Prolog, we convert each logical clause to a Prolog fact or rule. Within each clause, we use commutativity to move the negated terms to the right and the non-negated terms to the left (our example is already in this form). We then note that we can recast the disjunctions as implications:

$$\begin{aligned}
& (\text{student}(A) \leftarrow \neg(\neg\text{dorm\_resident}(A))) \\
& \qquad \wedge (\text{student}(A) \leftarrow \neg(\neg\text{takes}(A, B) \vee \neg\text{class}(B))) \\
\equiv\ & (\text{student}(A) \leftarrow \text{dorm\_resident}(A)) \\
& \qquad \wedge (\text{student}(A) \leftarrow (\text{takes}(A, B) \wedge \text{class}(B)))
\end{aligned}$$

These are Horn clauses. The translation to Prolog is trivial:

```
student(A) :- dorm_resident(A).
student(A) :- takes(A, B), class(B).
```

12.3.2 Limitations

We claimed at the beginning of Section 12.1 that Horn clauses could be used to capture most, though not all, of first-order predicate calculus. So what is it missing? What can go wrong in the translation? The answer has to do with the number of non-negated terms in each clause. If a clause has more than one, then if we attempt to cast it as an implication there will be a disjunction on the left-hand side of the ← symbol, something that isn't allowed in a Horn clause. Similarly, if we end up with no non-negated terms, then the result is a headless Horn clause, something that Prolog allows only as a query, not as an element of the database.

EXAMPLE 12.43
Disjunctive left-hand side

As an example of a disjunctive head, consider the statement "every living thing is an animal or a plant." In clausal form, we can capture this as

$$\text{animal}(X) \vee \text{plant}(X) \vee \neg\text{living}(X)$$

or equivalently

$$\text{animal}(X) \lor \text{plant}(X) \leftarrow \text{living}(X)$$

Because we are restricted to a single term on the left-hand side of a rule, the closest we can come to this in Prolog is

```
animal(X) :- living(X), \+(plant(X)).
plant(X) :- living(X), \+(animal(X)).
```

But this is not the same, because Prolog's \+ indicates inability to prove, not falsehood. ■

EXAMPLE 12.44
Empty left-hand side

As an example of an empty head, consider Fermat's last theorem (Example C-12.39). Abstracting out the math, we might write

$$\forall N[\text{big}(N) \rightarrow \neg(\exists A, \exists B, \exists C[\text{works}(A, B, C, N)])]$$

which becomes the following in clausal form:

$$\neg\text{big}(N) \lor \neg\text{works}(A, B, C, N)$$

We can couch this as a Prolog query:

```
?- big(N), works(A, B, C, N).
```

(a query that will never terminate), but we cannot express it as a fact or a rule. ■

EXAMPLE 12.45
Theorem proving as a search for contradiction

The careful reader may have noticed that facts are entered on the left-hand side of an (implied) Prolog :- sign:

```
rainy(rochester).
```

while queries are entered on the right:

```
?- rainy(rochester).
```

The former means

$$\text{rainy}(\text{rochester}) \leftarrow \text{true}$$

The latter means

$$\text{false} \leftarrow \text{rainy}(\text{rochester})$$

If we apply resolution to these two propositions, we end up with the contradiction

$$\text{false} \leftarrow \text{true}$$

This observation suggests a mechanism for automated theorem proving: if we are given a collection of axioms and we want to prove a theorem, we temporarily add the *negation* of the theorem to the database and then attempt, through a series of resolution operations, to obtain a contradiction. ■

12.3.3 Skolemization

EXAMPLE 12.46

Skolem constants

In Example C-12.41 we were able to translate a proposition from predicate calculus into clausal form without worrying about existential quantifiers. But what about a statement like this one:

$$\exists X[\mathsf{takes}(X, \mathsf{cs254}) \land \mathsf{class\_year}(X, 2)]$$

(There is at least one sophomore in cs254.) To get rid of the existential quantifier, we can introduce a *Skolem constant* x:

$$\mathsf{takes}(\mathsf{x}, \mathsf{cs254}), \mathsf{class\_year}(\mathsf{x}, 2)$$

The mathematical justification for this change is based on something called the *axiom of choice*; intuitively, we say that if there exists an X that makes the statement true, then we can simply pick one, name it x, and proceed. (If there does not exist an X that makes the statement true, then we can choose some arbitrary x, and the statement will still be false.) It is worth noting that Skolem constants are not necessarily distinct; it is quite possible, for example, for x to name the same student as some other constant y that represents a sophomore in `his201`. ■

EXAMPLE 12.47

Skolem functions

Sometimes we can replace an existentially quantified variable with an arbitrary constant x. Often, however, we are constrained by some surrounding universal quantifier. Consider the following example:

$$\forall X[\neg\mathsf{dorm\_resident}(X) \lor \exists A[\mathsf{campus\_address\_of}(X, A)]]$$

(Every dorm resident has a campus address.) To get rid of the existential quantifier, we must choose an address for X. Since we don't know who X is (this is a general statement about all dorm residents), we must choose an address that *depends on X*:

$$\forall X[\neg\mathsf{dorm\_resident}(X) \lor \mathsf{campus\_address\_of}(X, \mathsf{f}(X))]$$

Here f is a *Skolem function*. If we used a simple Skolem constant instead, we'd be saying that there exists some single address shared by all dorm residents. ■

EXAMPLE 12.48

Limitations of Skolemization

Whether Skolemization results in a clausal form that we can translate into Prolog depends on whether we need to know what the constant is. If we are using predicates `takes` and `class_year`, and we wish to assert as a fact that there is a sophomore in `cs254`, we can write

```
takes(the_distinguished_sophomore_in_254, cs254).
class_year(the_distinguished_sophomore_in_254, 2).
```

Similarly, we can assert that every dorm resident has a campus address by writing

```
campus_address_of(X, the_dorm_address_of(X)) :- dorm_resident(X).
```

Now we can search for classes with sophomores in them:

```
sophomore_class(C) :- takes(X, C), class_year(X, 2).
?- sophomore_class(C).
C = cs254
```

and we can search for people with campus addresses:

```
has_campus_address(X) :- campus_address_of(X, Y).
dorm_resident(li_ying).
?- has_campus_address(X).
X = li_ying
```

Unfortunately, we won't be able to identify a sophomore in `cs254` by name, nor will we be able to identify the address of `li_ying`. ■

✓ CHECK YOUR UNDERSTANDING

15. Define the notion of *clausal form* in predicate calculus.

16. Outline the procedure to convert an arbitrary predicate calculus statement into clausal form.

17. Characterize the statements in clausal form that cannot be captured in Prolog.

18. What is *Skolemization*? Explain the difference between Skolem constants and Skolem functions.

19. Under what circumstances may Skolemization fail to produce a clausal form that can be captured usefully in Prolog?

12 Logic Languages

12.6 Exercises

12.19 Restate the following Prolog rule in predicate calculus, using appropriate quantifiers:

```
sibling(X, Y) :- mother(M, X), mother(M, Y),
                 father(F, X), father(F, Y).
```

12.20 Consider the following statement in predicate calculus:

$$\mathsf{empty\_class}(C) \leftarrow \neg\exists X[\mathsf{takes}(X, C)]$$

(a) Translate this statement to clausal form.

(b) Can you translate the statement into Prolog? Does it make a difference whether you're allowed to use `\+`?

(c) How about the following:

$$\mathsf{takes\_everything}(X) \leftarrow \forall C[\mathsf{takes}(X, C)]$$

Can this be expressed in Prolog?

12.21 Consider the seemingly contradictory statement

$$\neg\mathsf{foo(X)} \rightarrow \mathsf{foo(X)}$$

Convert this statement to clausal form, and then translate into Prolog. Explain what will happen if you ask

```
?- foo(bar).
```

Now consider the straightforward translation, without the intermediate conversion to clausal form:

```
foo(X) :- \+(foo(X)).
```

Now explain what will happen if you ask

```
?- foo(bar).
```

12 Logic Languages

12.7 Explorations

12.27 In Section C-12.3.1 we translated propositions into *conjunctive normal form*: the AND of a collection of ORs. One can also translate propositions into *disjunctive normal form*: the OR of a collection of ANDs. Does disjunctive normal form have any useful properties? What other normal forms exist in mathematical logic? What are their uses?

12.28 With all the different ways to express the same proposition in predicate calculus, is there any useful notion of a "simplest" form? Is it possible, for example, to find, among all equivalent propositions, the one with the smallest number of symbols? How difficult is this task?

12.29 *Satisfiability* is the canonical NP-complete problem. Given a formula in propositional logic (no predicates or quantifiers), it asks whether there exists an assignment of truth values to variables that makes the overall proposition true. Can we use Prolog to solve the satisfiability problem? If not, why not? If so, given that it has to take exponential time, how can we hope to solve problems full of predicates and quantifiers quickly?

12.30 Suppose we had a form of "constructive negation" in Prolog that allowed us to capture information of the form $\forall X[\neg \mathsf{P}(X)]$. What might such a feature look like? What would be its implications for the Prolog search strategy? What portions of predicate calculus (if any) would still be inexpressible?

13 Concurrency

13.5 Message Passing

While shared-memory concurrent programming is common on small-scale multicore and multiprocessor machines, most programs that run on clusters, supercomputers, or geographically distributed machines are currently based on messages. In Sections C-13.5.1 through C-13.5.3 we consider three principal issues in message-based computing: naming, sending, and receiving. In Section C-13.5.4 we look more closely at one particular combination of send and receive semantics, namely remote procedure call. Most of our examples will be drawn from the Ada, Erlang, Go, and Rust programming languages, the Java network library, and the MPI library package.

13.5.1 Naming Communication Partners

EXAMPLE 13.54 Naming processes, ports, and entries

To send or receive a message, one must generally specify where to send it to, or where to receive it from: communication partners need names for (or references to) one another. Names may refer directly to a thread or process. Alternatively, they may refer to an *entry* or *port* of a module, or to some sort of *socket* or *channel* abstraction. We illustrate these options in Figure C-13.19. ■

The first naming option—addressing messages to processes—appears in Hoare's original CSP (Communicating Sequential Processes) [Hoa78], an influential proposal for simple communication mechanisms. It also appears in Erlang and in MPI. Each MPI process has a unique `id` (an integer), and each send or receive operation specifies the `id` of the communication partner. MPI implementations are required to be reentrant; a process can safely be divided into multiple threads, each of which can send or receive messages on the process's behalf.

EXAMPLE 13.55 `entry` calls in Ada

The second naming option—addressing messages to ports—appears in Ada. An Ada *entry call* of the form `t.foo(args)` sends a message to the `entry` named `foo` in task (thread) `t` (`t` may be either a task name or the name of a variable whose value is a pointer to a task). As we saw in Section 13.2.3, an Ada task resembles

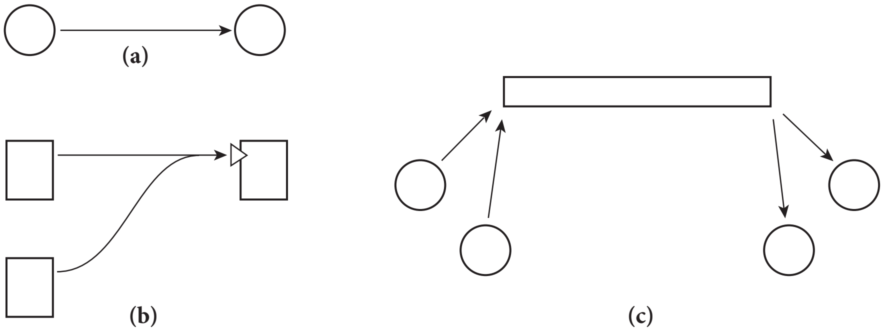


Figure 13.19 **Three common schemes to name communication partners.** In (a), processes name each other explicitly. In (b), senders name an *input port* of a receiver. The port may be called an *entry* or an *operation*. The receiver is typically a module with one or more threads inside. In (c), senders and receivers both name an independent *channel* abstraction, which may be called a *connection* or a *mailbox*.

a module; its entries resemble subroutine headers nested directly inside the task. A task receives a message that has been sent to one of its entries by executing an `accept` statement (to be discussed in Section C-13.5.3). Every `entry` belongs to exactly one task; all messages sent to the same `entry` must be received by that one task. ■

EXAMPLE 13.56
Channels in Go

The third naming option—addressing messages to channels—appears in Go, Occam, and Rust. (Though their concurrency features are loosely based on CSP, both Go and Occam differ from Hoare's proposal in several concrete ways, including the use of channels.) Channel declarations in Go are supported with the `chan` type constructor:

```
var c1 chan int
```

This code declares `c1` to be an (initially `nil`) reference to a channel. A channel value can be created with the built-in function `make`:

```
c1 = make(chan int)
```

Typically the declaration and initialization appear together:

```
var c1 = make(chan int)
```

Here Go infers the type of `c1` from the initialization expression.

To send a message on a channel, a thread uses the binary "arrow" operator `<-` with a channel variable on the left and a message on the right:

```
c1 <- 3
```

To receive, it uses `<-` as a unary operator, with the channel on the right:

```
my_int = <-c1
```

To indicate that no further messages will be forthcoming, a thread can *close* a channel. A receiving thread can check for this possibility by assigning a receive expression into a pair, the second element of which is a Boolean:

```
my_int, ok = <-c1
if (ok) {
    // use my_int ...
```

EXAMPLE 13.57
Remote invocation in Go

For the common idiom in which a server thread is willing to accept requests from any of many possible client threads, each request message can include a reference to the channel on which to send a response:

```
type request struct {
    name string
    reply_to chan string
}
...
// Assume a server thread is listening on chan 'service'
...
var c = make(chan string, 1)    // create channel for response
service <- request{"Alice", c}  // send look-up request for Alice
println(<-c)                    // receive response on c
```

EXAMPLE 13.58
Channels in Rust

Rust channels, unlike those of Go and Occam, have only a single receiver. More precisely, the receiving *end* of a channel is *owned* by a single thread. The sending end, by default, has a single owner as well, but unlike the receiving end it can be *cloned*:

```
let (tx, rx) = mpsc::channel();     // new channel
    // tx is the sending end; rx is the receiving end

for t in 0..2 {     // execute twice (for t = 0 and t = 1)
    let txc = tx.clone();
    thread::spawn(move || {
        txc.send(123 + t).unwrap();
    });
}

let v1 = rx.recv().unwrap();
let v2 = rx.recv().unwrap();
println!("{} {}", v1, v2);      // prints 123 124 or 124 123
```

Here `recv` returns a `Result<T>`—a datatype that can be either a previously sent value or a `RecvError<T>`, where `T` in this case is the integer type. The call to `unwrap` returns the value or raises a fatal error. The call to `send`, likewise, returns a `Result<()>`—an empty value or a `SendError<T>`; the `unwrap` serves simply to catch the error.

As might be expected in Rust's type system (Section 8.5.5), sending a value on a channel transfers *ownership* from the sending thread to the receiving. The following will not compile:

```
let s = String::from("boo!");
tx.send(s).unwrap();
println!("{}", s);             // not allowed!
```

The error message from the compiler explains that type `String` does not implement the `Copy` trait, so `v` cannot be accessed after the `send`. A `send` of a (trivially copied) integer would work just fine, as would a `send` of `s.clone()`.

Internet Messaging

Java's standard `java.net` library provides two styles of message passing, corresponding to the UDP and TCP Internet protocols. UDP is the simpler of the two. It is a *datagram* protocol, meaning that each message is sent to its destination independently and unreliably. The network software will attempt to deliver it, but makes no guarantees. Moreover two messages sent to the same destination (assuming they both arrive) may arrive in either order. UDP messages use port-based naming (Figure C-13.19b): each message is sent to a specific Internet protocol (IP) address and *port number*.[1] The TCP protocol also uses port-based naming, but only for the purpose of establishing *connections* (Figure C-13.19c), which it then uses for all subsequent communication. Connections deliver messages reliably and in order.

EXAMPLE 13.59
Datagram messages in Java

To send or receive UDP messages, a Java thread must create a *datagram socket*:

```
DatagramSocket mySocket = new DatagramSocket(portId);
```

The parameter of the `DatagramSocket` constructor is optional; if it is not specified, the operating system will choose an available port. Typically servers specify a port and clients allow the OS to choose. To send a UDP message, a thread says

```
DatagramPacket myMsg = new DatagramPacket(buf, len, addr, port);
...  // initialize message
mySocket.send(myMsg);
```

The parameters to the `DatagramPacket` constructor specify an array of bytes `buf`, its length `len`, and the Internet address and port of the receiver. Receiving is symmetric:

[1] Every publicly visible machine on the Internet has its own unique address. Though a transition to 128-bit addresses has been underway for some time, many sites still use 32-bit integers, usually printed as four period-separated fields (e.g., 192.5.54.209). Internet name servers translate symbolic names (e.g., `gate.cs.rochester.edu`) into numeric addresses. Port numbers are also integers, but are local to a given Internet address. Ports 1024 through 4999 are generally available for application programs; larger and smaller numbers are reserved for servers.

```
mySocket.receive(myMsg);
...  // parse content of myMsg
```

EXAMPLE 13.60
Connection-based messages in Java

For TCP communication, a server typically “listens” on a port to which clients send requests to establish a connection:

```
ServerSocket myServerSocket = new ServerSocket(portId);
Socket clientConnection = myServerSocket.accept();
```

The accept operation blocks until the server receives a connection request from a client. Typically a server will immediately fork a new thread to communicate with the client; the parent thread loops back to wait for another connection with accept.

A client sends a connection request by passing the server’s symbolic name and port number to the Socket constructor:

```
Socket serverConnection = new Socket(hostName, portId);
```

Once a connection has been created, a client and server in Java typically call methods of the Socket class to create input and output streams, which support all of the standard Java mechanisms for text I/O (Section C-8.7.3):

```
BufferedReader in = new BufferedReader(
    new InputStreamReader(clientConnection.getInputStream()));
PrintStream out =
    new PrintStream(clientConnection.getOutputStream());
// This is in the server; the client would make streams out
// of serverConnection.
...
String s = in.readLine();
out.println("Hi, Mom\n");
```

Among all the message-passing mechanisms we have considered, datagrams are the only one that does not provide some sort of *ordering* constraint. In general, most message-passing systems guarantee that messages sent over the same “communication path” arrive in order. When naming processes explicitly, a path links a single sender to a single receiver. All messages from that sender to that receiver arrive in the order sent. When naming ports, a path links an arbitrary number of senders to a single receiver. Messages that arrive at a port in a given order will be seen by receivers in that order. Note, however, that while messages from the same sender will arrive at a port in order, messages from *different* senders may arrive in arbitrary orders.[2] When naming channels, a path links all the senders that can

[2] Suppose, for example, that process A sends a message to port p of process B, and then sends a message to process C, while process C first receives the message from A and then sends its own message to port p of B. If messages are sent over a network with internal delays, and if A is allowed

use the channel to all the receivers that can use it. A Java TCP connection has a single OS process at each end, but there may be many threads inside, each of which can use its process's end of the connection. The connection functions as a queue: send (enqueue) and receive (dequeue) operations are ordered, so that everything is received in the order it was sent.

13.5.2 Sending

One of the most important issues to be addressed when designing a send operation is the extent to which it may block the caller: once a thread has initiated a send operation, when is it allowed to continue execution? Blocking can serve at least three purposes:

Resource management: A sending thread should not modify outgoing data until the underlying system has copied the old values to a safe location. Most systems block the sender until a point at which it can safely modify its data, without danger of corrupting the outgoing message.

Failure semantics: Particularly when communicating over a long-distance network, message passing is more error-prone than most other aspects of computing. Many systems block a sender until they are able to guarantee that the message will be delivered without error.

Return parameters: In many cases a message constitutes a *request*, for which a *reply* is expected. Many systems block a sender until a reply has been received.

When deciding how long to block, we must consider synchronization semantics, buffering requirements, and the reporting of run-time errors.

Synchronization Semantics

On its way from a sender to a receiver, a message may pass through many intermediate steps, particularly if traversing the Internet. It first descends through several layers of software on the sender's machine, then through a potentially large number of intermediate machines, and finally up through several layers of software on the receiver's machine. We could imagine unblocking the sender after any of these steps, but most of the options would be indistinguishable in terms of user-level program behavior. If we assume for the moment that a message-passing system can always find buffer space to hold an outgoing message, then our three rationales for delay suggest three principal semantic options:

EXAMPLE 13.61
Three main options for send semantics

No-wait send: The sender does not block for more than a small, bounded period of time. The message-passing implementation copies the message to a safe location and takes responsibility for its delivery.

to send its message to C before its first message has reached port p, then it is possible for B to hear from C before it hears from A. This apparent reversal of ordering could easily happen on the Internet, for example, if the message from A to B traverses a satellite link, while the messages from A to C and from C to B use ocean-floor fiber-optic cables.

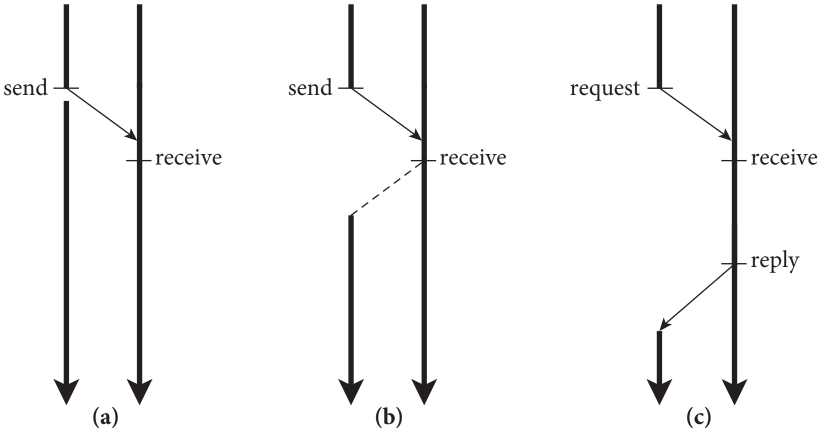


Figure 13.20 **Synchronization semantics for the** send **operation: no-wait** send **(a), synchronization** send **(b), and remote-invocation** send **(c).** In each diagram we have assumed that the original message arrives before the receiver executes its receive operation; this need not in general be the case.

Synchronization send: The sender waits until its message has been received.

Remote-invocation send: The sender waits until it receives a reply.

These three alternatives are illustrated in Figure C-13.20.

No-wait send appears in Erlang and its successor Elixir, in Rust, and in the Java Internet library. Synchronization send appears in Occam and, by default, in Go. (If a Go channel is declared with an explicit *buffering capacity*, however, no-wait send is used.) Remote-invocation send appears in Ada and in Occam. MPI provides an implementation-oriented hybrid of no-wait send and synchronization send: a send operation blocks until the data in the outgoing message can safely be modified. In implementations that do their own internal buffering, this rule amounts to no-wait send. In other implementations, it amounts to synchronization send. The programmer has the option, if desired, to insist on no-wait send or synchronization send; performance may suffer on some systems if the request is different from the default.

Buffering

In practice, of course, no message-passing system can provide a version of send that never waits (unless of course it simply throws some messages away). If we imagine a thread that sits in a loop sending messages to a thread that never receives them, we quickly see that unlimited amounts of buffer space would be required. At some point, any implementation must be prepared to block an overactive sender, to keep it from overwhelming the system. Such blocking is a form of *backpressure*. Milder backpressure can also be applied by reducing a thread's scheduling priority or by increasing the (still bounded) delay before a "no-wait" send returns.

EXAMPLE 13.62
Buffering-dependent deadlock

For any fixed amount of buffer space, it is possible to design a program that requires a larger amount of space to run correctly. Imagine, for example, that the message-passing system is able to buffer n messages on a given communication path. Now imagine a program in which A sends $n + 1$ messages to B, followed by one message to C. C then sends one message to B, on a different communication path. Finally, B insists on receiving the message from C before receiving the messages from A. If A blocks after message n, implementation-dependent deadlock will result. The best that an implementation can do is to provide a sufficiently large amount of space that realistic applications are unlikely to find the limit to be a problem. ■

For synchronization `send` and remote-invocation `send`, buffer space is not generally a problem: the total amount of space required for messages is bounded by the number of threads, and there are already likely to be limits on how many threads a program can create. A thread that sends a reply message can always be permitted to proceed: we know that we shall be able to reuse the buffer space quickly, because the thread that sent the request is already waiting for the reply.

Error Reporting

EXAMPLE 13.63
Acknowledgments

If the underlying message-passing system is unreliable, a language or library will typically employ *acknowledgment* messages to verify successful transmission (Figure C-13.21). If an acknowledgment is not received within a reasonable amount of time, the implementation will typically resend. If several attempts fail to elicit an acknowledgment, an error will be reported. ■

DESIGN & IMPLEMENTATION

13.11 The semantic impact of implementation issues

The inability to buffer unlimited amounts of data and, likewise, to report errors synchronously to a sender that has continued execution are only the most recent of many examples we have seen in which pragmatic implementation issues may restrict the language semantics available to the programmer. Other examples include limitations on the length of source lines or variable names (Section 2.1.1); limits on the memory available for data (whether global, stack, or heap allocated) and for recursive function evaluation (Section 3.2); the lack of ranges in `case` statement labels (Section 6.4.2); `in reverse` and constant step sizes for `for` loops (Section 6.5.1); limits on set universe size (to accommodate bit vectors—Section 8.4); limited procedure nesting (to accommodate displays—Section 9.1); the pointer-only restriction on opaque exports in Modula-2 (Section 10.2.1); and the lack of nested threads or of unrestricted arms on a `cobegin` statement (to avoid the need for cactus stacks—Section 9.5.1). Some of these limitations are reflected in the formal semantics of the language. Others (generally those that vary most from one implementation to another) restrict the set of semantically valid programs that the system will run correctly.

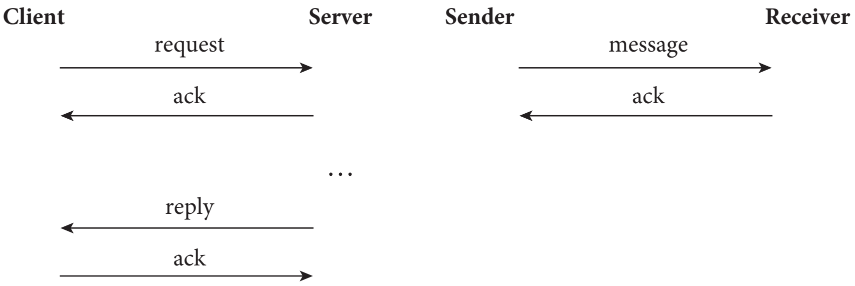


Figure 13.21 Acknowledgment messages for error detection. In the absence of piggy-backing, remote-invocation send (left) may require four underlying messages; synchronization send (right) may require two.

As long as the sender of a message is blocked, errors that occur in attempting to deliver a message can be reflected back as exceptions, or as status information in result parameters or global variables. Once a sender has continued, there is no obvious way in which to report any problems that arise. Like limits on message buffering, this dilemma poses semantic problems for no-wait send. For UDP, the solution is to state that messages are unreliable: if something goes wrong, the message is simply lost, silently. For TCP, the "solution" is to state that only "catastrophic" errors will cause a message to be lost, in which case the connection will become unusable and future calls will fail immediately. An even more drastic approach was taken in the original version of MPI: certain implementation-specific errors could be detected and handled at run time, but in general if a message could not be delivered then the program as a whole was considered to have failed. Newer versions of MPI provide a richer set of error-reporting facilities that can be used, with some effort, to build fault-tolerant programs.

Emulation of Alternatives

All three varieties of send can be emulated by the others. To obtain the effect of remote-invocation send, a thread can follow a no-wait send of a request with a receive of the reply, as we saw in Example C-13.57. Similar code will allow us to emulate remote-invocation send using synchronization send. To obtain the effect of synchronization send, a thread can follow a no-wait send with a receive of a high-level acknowledgment, which the receiver will send immediately upon receipt of the original message. To obtain the effect of synchronization send using remote-invocation send, a thread that receives a request can simply reply immediately, with no return parameters.

To obtain the effect of no-wait send using synchronization send or remote-invocation send, we must interpose a buffer process (the message-passing analogue of our shared-memory bounded buffer) that replies immediately to "senders" or "receivers" whenever possible. The space available in the buffer process makes explicit the resource limitations that are always present below the surface in implementations of no-wait send.

Syntax and Language Integration

In the emulation examples above, our hypothetical syntax assumed a library-based implementation of message passing. Because send, receive, accept, and so on are ordinary subroutines in such an implementation, they usually take a fixed, static number of parameters, two of which typically specify the location and size of the message to be sent. To send a message containing values held in more than one program variable, the programmer may need to explicitly *gather*, or *marshal*, those values into the fields of a record. On the receiving end, the programmer may then need to *scatter* (*unmarshal*) the values back into program variables. By contrast, a concurrent programming language can provide message-passing operations whose "argument" lists can include an arbitrary number of values to be sent. Moreover, the compiler can arrange to perform type checking on those values, using techniques similar to those employed for subroutine linkage across compilation units (to be described in Section 15.6.2). Finally, as we shall see in Section C-13.5.3, an explicitly concurrent language can employ non-procedure-call syntax—for example, to couple a remote-invocation accept and reply in such a way that the reply doesn't have to explicitly identify the accept to which it corresponds.

DESIGN & IMPLEMENTATION

13.12 Emulation and efficiency

Unfortunately, user-level emulations of alternative send semantics are seldom as efficient as optimized implementations using the underlying primitives. Suppose for example that we wish to use remote-invocation send to emulate synchronization send. Suppose further that our implementation of remote-invocation send is built on top of network software that needs acknowledgments to verify message delivery. After sending a reply, the server's run-time system will wait for an acknowledgment from the client. If a server thread can work for an arbitrary amount of time before sending a reply, then the run-time system will need to send separate acknowledgments for the request and the reply. If a programmer uses this implementation of remote-invocation send to emulate synchronization send, then the underlying network may end up transmitting a total of four messages (more if there are any transmission errors). By contrast, a "native" implementation of synchronization send would require only two underlying messages. In some cases the run-time system for remote-invocation send may be able to delay transmission of the first acknowledgment long enough to "piggy-back" it on the subsequent reply if there is one; in this case an emulation of synchronization send may transmit three underlying messages instead of only two. We consider the efficiency of emulations further in Exercise C-13.36 and Exploration C-13.55.

13.5.3 Receiving

Probably the most important dimension on which to categorize mechanisms for receiving messages is the distinction between explicit receive operations and the *implicit* receipt described in Section 13.2.3. Among the languages and systems we have been using as examples, none provides implicit receipt, but it appears in a variety of research languages, and in some of the RPC systems we will consider in Section C-13.5.4).

With implicit receipt, every message that arrives at a given port (or over a given channel) will create a new thread of control, subject to resource limitations (any implementation will have to stall incoming requests when the number of threads grows too large). With explicit receipt, a message will be queued until some already-existing thread indicates a willingness to receive it. At any given point in time there may be a potentially large number of messages waiting to be received. Most languages and libraries with explicit receipt allow a thread to exercise some sort of *selectivity* with respect to which messages it wants to consider.

In MPI, every message includes the id of the process that sent it, together with an integer *tag* specified by the sender. A receive operation specifies a desired sender id and message tag. Only matching messages will be received. In many cases receivers specify "wild cards" for the sender id and/or message tag, allowing any of a variety of messages to be received. Special versions of receive also allow a process to test (without blocking) to see if a message of a particular type is currently available (this operation is known as *polling*), or to "time out" and continue if a matching message cannot be received within a specified interval of time.

Because they are languages instead of library packages, Ada, Erlang/Elixir, Go, and Occam are able to use special, non-procedure-call syntax for selective message receipt. Moreover because messages are built into the naming and typing system, these languages are able to receive selectively on the basis of port/channel names and parameters, rather than the more primitive notion of tags. In all four languages, the selective receive construct is a special form of *guarded command*, as described in Section C-6.7.

EXAMPLE 13.64
Bounded buffer in Ada 83

Figure C-13.22 contains code for a bounded buffer in Ada 83. Here an active "manager" thread executes a select statement inside a loop. (Recall that it is also possible to write a bounded buffer in Ada using *protected objects*, without a manager thread, as described in Section 13.4.3.) The Ada accept statement receives the in and in out parameters (Section 9.3.1) of a remote invocation request. At the matching end, accept returns the in out and out parameters as a reply message. A client task would communicate with the bounded buffer using an *entry call*:

```
-- producer:                    -- consumer:
buffer.insert(3);               buffer.remove(x);
```

The select statement in our buffer example has two arms. The first arm may be selected when the buffer is not full and there is an available insert request; the second arm may be selected when the buffer is not empty and there is an

```
task buffer is
    entry insert(d : in bdata);
    entry remove(d : out bdata);
end buffer;

task body buffer is
    SIZE : constant integer := 10;
    subtype index is integer range 1..SIZE;
    buf : array (index) of bdata;
    next_empty, next_full : index := 1;
    full_slots : integer range 0..SIZE := 0;
begin
    loop
        select
          when full_slots < SIZE =>
            accept insert(d : in bdata) do
                buf(next_empty) := d;
            end;
            next_empty := next_empty mod SIZE + 1;
            full_slots := full_slots + 1;
        or
          when full_slots > 0 =>
            accept remove(d : out bdata) do
                d := buf(next_full);
            end;
            next_full := next_full mod SIZE + 1;
            full_slots := full_slots - 1;
        end select;
    end loop;
end buffer;
```

Figure 13.22 Bounded buffer in Ada, with an explicit manager task.

available `remove` request. Selection among arms is a two-step process: first the guards (`when` expressions) are evaluated, then for any that are true the subsequent `accept` statements are considered to see if a message is available. (The guard in front of an `accept` is optional; if missing it behaves as `when true =>`.) If both of the guards in our example are true (the buffer is partly full) and both kinds of messages are available, then either arm of the statement may be executed, at the discretion of the implementation. (For a discussion of issues of *fairness* in the choice among true guards, see Sidebar C-6.10.)

EXAMPLE 13.65
Timeout and distributed termination

Every `select` statement must have at least one arm beginning with `accept` (and optionally `when`). In addition, it may have three other types of arms:

```
when condition => delay how_long
    other_statements
...
or when condition => terminate
...
else ...
```

A `delay` arm may be selected if no other arm becomes selectable within *how_long* seconds. (Ada implementations are required to support delays as long as 1 day or as short as 20 ms.) A `terminate` arm may be selected only if all potential communication partners have already terminated or are likewise stuck in `select` statements with `terminate` arms. Selection of the arm causes the task that was executing the `select` statement to terminate. An `else` arm, if present, will be selected when none of the guards are true or when no `accept` statement can be executed immediately. A `select` statement with an `else` arm is not permitted to have any `delay` arms. In practice, one would probably want to include a `terminate` arm in the `select` statement of a manager-style bounded buffer.

EXAMPLE 13.66
Bounded buffer in Go

In Go, a bounded buffer is trivial: it's just a buffered channel:

```
type bdata struct {
    n int   // or whatever
}
var buffer = make(chan bdata, 10)   // space for ten items of type bdata
...
buffer <- bdata{3}                  // insert
...
my_int = (<-buffer).n               // remove
```

To illustrate language features, we can also build a bounded buffer with an explicit thread, an array, and a pair of default (unbuffered) channels, in a manner similar to the Ada example of Figure C-13.22, but with synchronization send instead of remote invocation. Code for this alternative appears in Figure C-13.23. Unlike built-in buffered channels, it could easily be augmented to support functionality like priority-based (as opposed to FIFO) queueing, or methods to clear the buffer or to query the number of messages currently queued. To use the basic `insert`/`remove` operations, we might write:

```
var b = make_buffer()
...
b.insert(bdata{3})                  // insert
...
my_int = b.remove().n               // remove
```

As in the Ada example, requests are processed by an active manager thread (called a "goroutine" in Go), here started with the `go` command. The `select` statement in Go does not support explicit guards; we have achieved a similar effect in Figure C-13.23 by setting the `ic` and `rc` channels to `nil` when they should not

```
type buffer struct {
    full_slots, next_full, next_empty int
    buf [SIZE]bdata
    insert_c chan bdata
    remove_c chan chan bdata
}
func manager(b *buffer) {
    var ic chan bdata = b.insert_c
    var rc chan chan bdata = nil
    for {
        select {  // at least one of ic and rc will always be non-nil
          case d := <-ic:       // := means "declare and initialize"
            b.buf[b.next_empty] = d
            b.next_empty = (b.next_empty + 1) % SIZE
            b.full_slots++
            rc = b.remove_c     // there is definitely data to remove
            if b.full_slots == SIZE { ic = nil }
          case c := <-rc:
            c <- b.buf[b.next_full]
            b.next_full = (b.next_full + 1) % SIZE
            b.full_slots--
            ic = b.insert_c     // there is definitely space to fill
            if b.full_slots == 0 { rc = nil }
        }
    }
}
func make_buffer() (b *buffer) {    // return value has name 'b'
    b = new(buffer)
    b.full_slots = 0
    b.next_full = 0
    b.next_empty = 0
    b.insert_c = make(chan bdata)
    b.remove_c = make(chan chan bdata)
    go manager(b)               // create active manager thread
    return
}
func (b *buffer) insert(e bdata) {
    b.insert_c <- e             // send data to manager
}
func (b *buffer) remove() bdata {
    var c = make(chan bdata)
    b.remove_c <- c             // send temporary channel to manager
    return <-c                  // receive and return response
}
```

Figure 13.23 Bounded buffer with an explicit manager thread in Go. The `insert` and `remove` functions serve as methods of buffer `b`. Note that in the absence of additional functionality (not shown), this code would better be replaced by trivial use of a buffered channel with capacity `SIZE`. Also, if using this version, we would probably want a way to terminate the manager thread when the buffer is no longer needed.

```
buffer(Max, Free, Q) ->
    receive
        {insert, D, Client} when Free > 0 ->
            Client ! ok,                            % send ack
            buffer(Max, Free-1, queue:in(D, Q));    % enqueue
        {remove, Client} when Free < Max ->
            {{value, D}, NewQ} = queue:out(Q),      % dequeue
            Client ! D,                             % send element
            buffer(Max, Free+1, NewQ)
    end.
```

Figure 13.24 **Bounded buffer in Erlang.** Variables (names that can be instantiated with a value) begin with a capital letter; constants begin with a lower-case letter. Queue operations (`in`, `out`) are provided by the standard Erlang library. Typing is dynamic. The send operator (`!`) is as in CSP and Occam. Each clause of the receive ends with a tail recursive call.

be selected. Because we have used synchronization send—channels `insert_c` and `remove_c` have zero capacity—there is an asymmetry between the handling of `insert` and `remove` requests: the former need only send the manager data; the latter must send a channel reference and then wait for the manager to send the data back. ■

EXAMPLE 13.67
Bounded buffer in Erlang

In Erlang, which uses no-wait send, one might at first expect asymmetry similar to that of Figure C-13.23: a consumer would have to receive a reply from a bounded buffer, but a producer could simply send data. Such asymmetry would have a hidden flaw, however: because a process does not wait after sending, the producer could easily send more items than the buffer can hold, with the excess being buffered in the message system. If we want the buffer to truly be bounded, we must require the producer to wait for an acknowledgment. Code for the buffer appears in Figure C-13.24. Because Erlang is a functional language, we use tail recursion instead of iteration. Code for the producer and consumer looks like this:

```
-- producer:                    -- consumer:
Buffer ! {insert, X, self()},   Buffer ! {remove, self()},
receive ok -> [] end.           receive X -> [] end.
```

The exclamation point (`!`), borrowed from CSP, is used to send a message. ■

EXAMPLE 13.68
Peeking at messages in Erlang

Several languages—Erlang among them—place the parameters of an incoming message within the scope of the guard condition, allowing a receiver to "peek inside" a message before deciding whether to receive it. In Erlang, we can say

```
receive
    {insert, D} when D rem 2 == 1 ->    % accept only odd numbers
```

The ability to peek implies that the content of incoming messages must be visible to the language run-time system. An Erlang implementation must therefore be prepared to accept (and buffer) an arbitrary number of messages; it cannot rely on the operating system or other underlying software to provide the buffering for it.

Moreover the fact that buffer space can never be truly unlimited means that guards and scheduling expressions will be unable to see messages whose delivery has been delayed by backpressure. ■

13.5.4 Remote Procedure Call

Any of the three principal forms of send (no-wait, synchronization, remote-invocation) can be paired with either of the principal forms of receive (explicit or implicit). The combination of remote-invocation send with explicit receipt (e.g., as in Ada) is sometimes known as *rendezvous*. The combination of remote-invocation send with implicit receipt is usually known as *remote procedure call*. RPC is available in several concurrent languages, and is also supported on many systems by augmenting a sequential language with a *stub compiler*. The stub compiler is independent of the language's regular compiler. It accepts as input a formal description of the subroutines that are to be called remotely. The description is roughly equivalent to the subroutine headers and declarations of the types of all parameters. Based on this input the stub compiler generates source code for *client* and *server stubs*. A client stub for a given subroutine marshals request parameters and an indication of the desired operation into a message buffer, sends the message to the server, waits for a reply message, and unmarshals that message into result parameters. A server stub takes a message buffer as parameter, unmarshals request parameters, calls the appropriate local subroutine, marshals return parameters into a reply message, and sends that message back to the appropriate client. Invocation of a client stub is relatively straightforward. Invocation of server stubs is discussed under "Implementation" below.

Semantics

A principal goal of most RPC systems is to make the remote nature of calls as *transparent* as possible; that is, to make remote calls look as much like local calls as possible [BN84]. In a stub compiler system, a client stub should have the same interface as the remote procedure for which it acts as proxy; the programmer should usually be able to call the routine without knowing or caring whether it is local or remote.

Several issues make it difficult to achieve transparency in practice:

Parameter modes: It is difficult to implement call-by-reference parameters across a network, since actual parameters will not be in the address space of the called routine. (Access to global variables is similarly difficult.)

Performance: There is no escaping the fact that remote procedures may take a long time to return. In the face of network delays, one cannot use them casually.

Failure semantics: Remote procedures are much more likely to fail than are local procedures. It is generally acceptable in the local case to assume that a called procedure will either run exactly once or else the entire program will fail. Such an assumption is overly restrictive in the remote case.

We can use value/result parameters in place of reference parameters so long as program correctness does not rely on the aliasing created by reference parameters. As noted in Section 9.3.1, Ada declares that a program is *erroneous* if it can tell the difference between pass-by-reference and pass-by-value/result implementations of `in out` parameters. If absolutely necessary, reference parameters and global variables can be implemented with message-passing thunks in a manner reminiscent of call-by-name parameters (Section C-9.3.2), but only at very high cost. As noted in Section 7.5, a few languages and systems perform deep copies of linked data structures passed to remote routines.

Performance differences between local and remote calls can be hidden only by artificially slowing down the local case. Such an option is clearly unacceptable.

Exactly-once failure semantics can be provided by aborting the caller in the event of failure or, in highly reliable systems, by delaying the caller until the operating system or language run-time system is able to rebuild the failed computation using information previously dumped to disk. (Failure recovery techniques are beyond the scope of this text.) An attractive alternative is to accept "at-most-once" semantics with notification of failure. The implementation retransmits requests for remote invocations as necessary in an attempt to recover from lost messages. It guarantees that retransmissions will never cause an invocation to happen more than once, but it admits that in the presence of communication failures the invocation may not happen at all. If the programming language provides exceptions then the implementation can use them to make communication failures look like any other kind of run-time error.

DESIGN & IMPLEMENTATION

13.13 Parameters to remote procedures

Ada's comparatively high-level semantics for parameter modes allows the same set of modes to be used for both subroutines and entries (rendezvous). An Ada compiler will generally pass a large argument to a subroutine by reference whenever possible, to avoid the expense of copying. If tasks are on separate nodes of a cluster, however, the compiler will generally pass the same argument to an entry by value-result.

A few concurrent languages provide parameter modes specifically designed with remote invocation in mind. In Emerald [BHJL07], for example, every parameter is a reference to an object. References to remote objects are implemented transparently via message passing. To minimize the frequency of such references, objects passed to remote procedures often *migrate* with the call: they are packaged with the request message, sent to the remote site (where they can be accessed locally), and returned to the caller in the reply. Emerald calls this *call by move*. In Hermes [SBG$^+$91] parameter passing is *destructive*, much like sending on a channel in Rust (Example C-13.58). Arguments become uninitialized from the caller's point of view, and can therefore migrate to a remote callee without danger of inducing remote references.

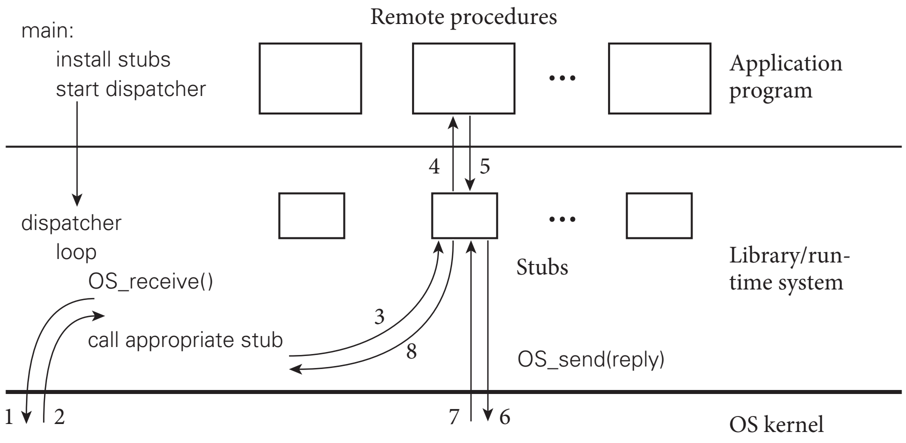


Figure 13.25 **Implementation of a remote procedure call server.** Application code initializes the RPC system by installing stubs generated by the stub compiler (not shown). It then calls into the run-time system to enable incoming calls. Depending on details of the particular system in use, the dispatcher may use the thread from the main program (in which case the call to start the dispatcher never returns), or it may create a pool of threads that handle incoming requests.

Implementation

At the level of the kernel interface, receive is usually an explicit operation. To make receive appear implicit to the application programmer, the code produced by an RPC stub compiler (or the run-time system of an RPC-based language) must bridge this explicit-to-implicit gap. The typical implementation resembles the thread-based event handling of Section 9.6.2. We describe it here in terms of stub compilers; in a concurrent language with implicit receipt the regular compiler does essentially the same work.

EXAMPLE 13.69

An RPC server system

Figure C-13.25 illustrates the layers of a typical RPC system. Code above the upper horizontal line is written by the application programmer. Code in the middle is a combination of library routines and code produced by the RPC stub compiler. To initialize the RPC system, the application makes a pair of calls into the run-time system. The first provides the system with pointers to the stub routines produced by the stub compiler; the second starts a *message dispatcher*. What happens after this second call depends on whether the server is concurrent and, if so, whether its program threads are implemented on top of one kernel thread or several.

In the simplest case—a single-threaded server on a single kernel thread—the dispatcher runs a loop that calls into the kernel to receive a message. When a message arrives, the dispatcher calls the appropriate RPC stub, which unmarshals request parameters and calls the appropriate application-level procedure. When that procedure returns, the stub marshals return parameters into a reply message, calls into the kernel to send the message back to the caller, and then returns to the dispatcher. ■

This simple organization works well so long as each remote request can be handled quickly, without ever needing to block. If remote requests must sometimes wait for user-level synchronization, then the server's process must manage a ready list of threads, as described in Section 13.2.4, but with the dispatcher integrated into the usual thread scheduler. When the current thread blocks (in application code), the scheduler/dispatcher will grab a new thread from the ready list. If the ready list is empty, the scheduler/dispatcher will call into the kernel to receive a message, fork a new user-level thread to handle it, and then continue to execute runnable threads until the list is empty again (each thread will terminate when it finishes handling its request).

In a multithreaded server, the call to start the dispatcher will generally ask the kernel to fork a "pool" of threads to service remote requests. Each of these threads will then perform the operations described in the previous paragraphs. In a language or library with a one–one correspondence between program threads and kernel threads, each will repeatedly receive a message from the kernel, call the appropriate stub, and loop back for another request. With a more general thread package, each kernel thread will run threads from the application's ready list until the list is empty, at which point it (the kernel thread) will call into the kernel for another message. So long as the number of runnable program threads is greater than or equal to the number of kernel threads, no new messages will be received. When the number of runnable program threads drops below the number of kernel threads, the extra kernel threads will call into the kernel, where they will block until requests arrive.

✓ CHECK YOUR UNDERSTANDING

50. Describe three ways in which processes or threads commonly name their communication partners.
51. What is a *datagram*?
52. Why, in general, might a `send` operation need to block?
53. What are the three principal synchronization options for the sender of a message? What are the tradeoffs among them?
54. What are *gather* and *scatter* operations in a message-passing program? What are *marshalling* and *unmarshalling*?
55. Describe the tradeoffs between *explicit* and *implicit* message receipt.
56. What is a *remote procedure call* (RPC)? What is a *stub compiler*?
57. What are the obstacles to *transparency* in an RPC system?
58. What is a *rendezvous*? How does it differ from a remote procedure call?
59. Explain the purpose of a `select` statement in Ada or Go.

60. What semantic and pragmatic challenges are introduced by the ability to "peek" inside messages before they are received?

13 Concurrency

13.7 Exercises

13.34 In Section 13.4.2 we cast monitors as a mechanism for synchronizing access to shared memory, and we described their implementation in terms of semaphores. It is also possible to think of a monitor as a module inhabited by a single thread, which accepts request messages from other threads, performs appropriate operations, and replies. Give the details of a monitor implementation consistent with this conceptual model. Be sure to include condition variables. (Hint: See the discussion of early reply in Section 13.2.3.)

13.35 Show how shared memory can be used to implement message passing. Specifically, choose a set of message-passing operations (e.g., no-wait `send` and explicit message receipt) and show how to implement them in your favorite shared-memory notation.

13.36 When implementing reliable messages on top of unreliable messages, a sender can wait for an acknowledgment message, and retransmit if it doesn't receive it within a bounded period of time. But how does the receiver know that its acknowledgment has been received? Why doesn't the sender have to acknowledge the acknowledgment (and the receiver acknowledge the acknowledgment of the acknowledgment . . .)? (For more information on the design of fast, reliable protocols, you might want to consult a text on computer networks [TFW21, PD21].)

13.37 Write a channel-based bounded buffer with an explicit manager thread in Rust, patterned after the Go version of Figure C-13.23. You will want to read up on the `select` macro of the `crossbeam_channel` crate.

13.38 While Go allows both *input* (receive) and *output* (send) guards on its `select` statements, Occam and CSP allow only input guards. The difference has to do with the fact that Go is designed for communication among threads in a single address space, while Occam and CSP were designed for a distributed environment. Why should this make a difference? Suppose you wished to add output guards to Occam. How would the implementation work? (Hint: For ideas, see the article by Bagrodia [Bag89].)

13.39 In Section C-13.5.3 we described the semantics of a `terminate` arm on an Ada `select` statement: this arm may be selected if and only if all potential communication partners have terminated, or are likewise stuck in `select` statements with `terminate` arms. Erlang and Occam have no similar facility, though the original CSP proposal does. How would you implement `terminate` arms in Ada? Why do you suppose they were left out of Erlang and Occam? (Hint: For ideas, see the work of Apt and Francez [Fra80, AF84].)

13 Concurrency

13.8 Explorations

13.55 Find out how message passing is implemented in some locally available concurrent language or library. Does this system provide no-wait `send`, synchronization `send`, remote-invocation `send`, or some related hybrid? If you wanted to emulate the other options using the one available, how expensive would emulation be, in terms of low-level operations performed by the underlying system? How would this overhead compare to what could be achieved on the same underlying system by a language or library that provided an optimized implementation of the other varieties of `send`?

13.56 Learn about Elixir, the Erlang successor due to José Valim. What are the principle differences between the two languages? How compatible are their implementations?

13.57 MPI provides extensive facilities for *collective communication*, in which there are more than two communicating parties. Examples include *multicast*, in which a message is sent simultaneously to a group of recipients; *scatter*, in which elements of an array-structured message are sent, one each, to a group of recipients; *gather*, in which an array-structured message is created, at the sole recipient, from elements provided by a group of senders; *all-to-all*, in which participants provide one element each of an array-structured message that is received by all; and *reduction*, in which messages from a group of senders are combined, using a commutative operator, into a result that is received by one or all. Learn more about both the semantics and the implementation of collective communication. What opportunities does it provide for optimizations that are difficult to implement at the application level?

13.58 Language designers and concurrency experts have argued for nearly 40 years over whether shared memory or message passing is a more appealing programming model. The argument is to a large extent subjective—and hence not subject to definitive settlement—but it includes substantive issues of fault containment, implementation efficiency, hardware requirements, and algorithmic expressiveness as well. Do a literature search on "shared memory versus message passing." How many papers do you find? Read a sampling of these and summarize their arguments. Do you find any of the positions particularly convincing? What do you think of the decision to include both options in Rust?

14 Scripting

14.3 Scripting the World Wide Web

Much of the content of the World Wide Web—particularly the content that is visible to search engines—is static: pages that seldom, if ever, change. But hypertext, the abstract notion on which the Web is based, was always conceived as a way to represent "the complex, the changing, and the indeterminate" [Nel65]. Much of the power of the Web today lies in its ability to deliver pages that move, play sounds, respond to user actions, or—perhaps most important—contain information created or formatted on demand, in response to the page-fetch request.

From a programming languages point of view, simple playback of recorded audio or video is not particularly interesting. We therefore focus our attention here on content that is generated on the fly by a program—a script—associated with an Internet URI (uniform resource identifier).[1] Suppose we type a URI into a browser on a client machine, and the browser sends a request to the appropriate web server. If the content is dynamically created, an obvious first question is: does the script that creates it run on the server or the client machine? These options are known as *server-side* and *client-side* web scripting, respectively.

Server-side scripts are typically used when the service provider wants to retain complete control over the content of the page, but can't (or doesn't want to) create the content in advance. Examples include the pages returned by search engines, Internet retailers, auction sites, and any organization that provides its clients with on-line access to personal accounts. Client-side scripts are typically used for tasks that don't need access to proprietary information, and are more efficient if executed on the client's machine. Examples include interactive animation, error-checking of fill-in forms, and a wide variety of other self-contained calculations.

[1] The term "URI" is often used interchangeably with "URL" (uniform resource locator), but the World Wide Web Consortium distinguishes between the two. All URIs are hierarchical (multipart) names. URLs are one kind of URIs; they use a naming scheme that indicates where to find the resource. Other URIs can use other naming schemes.

```
#!/usr/bin/perl

print "Content-type: text/html\n\n";
print "<!DOCTYPE html>\n";

print "<html lang=\"en\">\n";
$host = `hostname`; chop $host;
print "<head>\n";
print "<meta charset=\"utf-8\">\n";
print "<title>Status of ", $host, "</title>\n";
print "</head>\n<body>\n";
print "<h1>", $host, "</h1>\n";
print "<pre>\n", `uptime`, "\n", `who`;
print "</pre>\n</body>\n</html>\n";
```

Figure 14.14 **A simple CGI script in Perl.** If this script is named `status.perl`, and is installed in the server's `cgi-bin` directory, then a user anywhere on the Internet can obtain summary statistics and a list of users currently logged into the server by typing *hostname/cgi-bin/status.perl* into a browser window.

14.3.1 CGI Scripts

The original mechanism for server-side web scripting was the Common Gateway Interface (CGI). A CGI script is an executable program residing in a special directory known to the web server program. When a client requests the URI corresponding to such a program, the server executes the program and sends its output back to the client. Naturally, this output needs to be something that the browser will understand—typically HTML.

EXAMPLE 14.77

Remote monitoring with a CGI script

CGI scripts may be written in any language available on the server's machine, though Perl is particularly popular: its string-handling and "glue" mechanisms are ideally suited to generating HTML, and it was already widely available during the early years of the Web. As a simple if somewhat artificial example, suppose we would like to be able to monitor the status of a server machine shared by some community of users. The Perl script in Figure C-14.14 creates a web page titled by the name of the server machine, and containing the output of the `uptime` and `who` commands (two simple sources of status information). The script's initial `print` command produces an HTTP message header, indicating that what follows is HTML. Sample output from executing the script appears in Figure C-14.15. ■

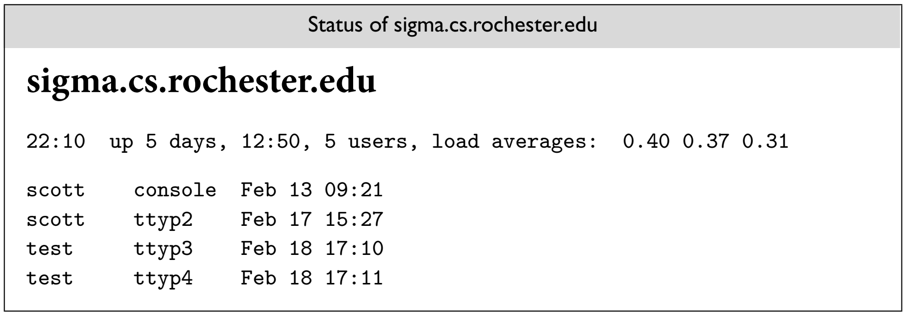
EXAMPLE 14.78

Adder web form with a CGI script

CGI scripts are commonly used to process on-line forms. A simple example appears in Figure C-14.16. The `form` element in the HTML file specifies the URI of the CGI script, which is invoked when the user hits the Submit button. Values previously entered into the `input` fields are passed to the script either as a trailing part of the URI (for a `get`-type form) or on the standard input stream (for a `post`-type form, shown here).[2] With either method, we can access the values using the

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Status of sigma.cs.rochester.edu</title>
</head>
<body>
<h1>sigma.cs.rochester.edu</h1>
<pre>
22:10  up 5 days, 12:50, 5 users, load averages: 0.40 0.37 0.31

scott    console  Feb 13 09:21
scott    ttyp2    Feb 17 15:27
test     ttyp3    Feb 18 17:10
test     ttyp4    Feb 18 17:11
</pre>
</body>
</html>
```



Status of sigma.cs.rochester.edu

sigma.cs.rochester.edu

```
22:10  up 5 days, 12:50, 5 users, load averages:  0.40 0.37 0.31

scott    console  Feb 13 09:21
scott    ttyp2    Feb 17 15:27
test     ttyp3    Feb 18 17:10
test     ttyp4    Feb 18 17:11
```

Figure 14.15 Sample output from the script of Figure C-14.14. HTML source appears at top; the rendered page is below.

`param` routine of the standard `CGI` Perl library, loaded at the beginning of our script. ■

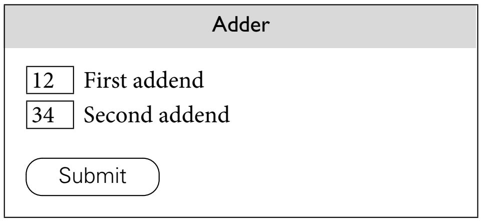
14.3.2 Embedded Server-Side Scripts

Though widely used, CGI scripts have several disadvantages:

- The web server must launch each script as a separate program, with potentially significant overhead (though a CGI script compiled to native code can be very fast once running).

[2] One typically uses `post` type forms for one-time requests. A `get` type form appears a little clumsier, because arguments are visibly embedded in the URI, but this gives it the advantage of repeatability: it can be "bookmarked" by client browsers.

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Adder</title></head>
<body>
<form action="/cgi-bin/add.perl" method="post">
<p><input name="argA" size=3>First addend<br>
   <input name="argB" size=3>Second addend</p>
<p><input type="submit"></p>
</form>
</body>
</html>
```



Adder

12 First addend

34 Second addend

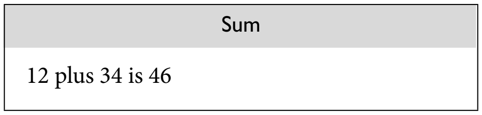
```
#!/usr/bin/perl

use CGI qw(:standard);      # provides access to CGI input fields
$argA = param("argA");  $argB = param("argB");  $sum = $argA + $argB;

print "Content-type: text/html\n\n";
print "<!DOCTYPE html>\n";

print "<html lang=\"en\">\n";
print "<head><meta charset=\"utf-8\"><title>Sum</title></head>\n<body>\n";
print "<p>$argA plus $argB is $sum</p>\n";
print "</body>\n</html>\n";
```

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Sum</title></head>
<body>
<p>12 plus 34 is 46</p>
</body>
</html>
```



Sum

12 plus 34 is 46

Figure 14.16 **An interactive CGI form.** Source for the original web page is shown at the upper left, with the rendered page to the right. The user has entered 12 and 34 in the text fields. When the Submit button is pressed, the client browser sends a request to the server for URI */cgi-bin/add.perl*. The values 12 and 13 are contained within the request. The Perl script, shown in the middle, uses these values to generate a new web page, shown in HTML at the bottom left, with the rendered page to the right.

- Because the server has little control over the behavior of a script, scripts must generally be installed in a trusted directory by trusted system administrators; they cannot reside in arbitrary locations as ordinary pages do.
- The name of the script appears in the URI, typically prefixed with the name of the trusted directory, so static and dynamic pages look different to end users.

```
<!DOCTYPE html>
<html lang="en">
<head>
<meta charset="utf-8">
<title>Status of <?php echo $host = chop(`hostname`) ?></title>
</head>
<body>
<h1><?php echo $host ?></h1>
<pre>
<?php echo `uptime`, "\n", `who` ?>
</pre>
</body>
</html>
```

Figure 14.17 A simple PHP script embedded in a web page. When served by a PHP-enabled host, this page performs the equivalent of the CGI script of Figure C-14.14.

- Each script must generate not only dynamic content but also the HTML tags that are needed to format and display it. This extra "boilerplate" makes scripts more difficult to write.

To address these disadvantages, most web servers provide a "module-loading" mechanism that allows interpreters for one or more scripting languages to be incorporated into the server itself. Scripts in the supported language(s) can then be embedded in "ordinary" web pages. The web server interprets such scripts directly, without launching an external program. It then replaces the scripts with the output they produce, before sending the page to the client. Clients have no way to even know that the scripts exist.

Embeddable server-side scripting languages include PHP, PowerShell (in Microsoft Active Server Pages), Ruby, Cold Fusion (from Macromedia Corp.), and Java (via "Servlets" in Java Server Pages). The most common of these is PHP. Though descended from Perl, PHP has been extensively customized for its target domain, with built-in support for (among other things) email and MIME encoding, all the standard Internet communication protocols, authentication and security, HTML and URI manipulation, and interaction with dozens of database systems.

EXAMPLE 14.79

Remote monitoring with a PHP script

The PHP equivalent of Figure C-14.14 appears in Figure C-14.17. Most of the text in this figure is standard HTML. PHP code is embedded between `<?php` and `?>` delimiters. These delimiters are not themselves HTML; rather, they indicate a *processing instruction* that needs to be executed by the PHP interpreter to generate replacement text. The "boilerplate" parts of the page can thus appear verbatim; they need not be generated by `print` (Perl) or `echo` (PHP) commands. Note that the separate script fragments are part of a single program. The `$host` variable, for example, is set in the first fragment and used again in the second. ■

EXAMPLE 14.80

A fragmented PHP script

PHP scripts can even be broken into fragments in the middle of structured statements. Figure C-14.18 contains a script in which `if` and `for` statements span fragments. In effect, the HTML text between the end of one script fragment and the beginning of the next behaves as if it had been output by an `echo` command.

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>20 numbers</title></head>
<body>
<p>
<?php
         for ($i = 0; $i < 20; $i++) {
             if ($i % 2) { ?>
<b><?php
                 echo " $i"; ?>
</b><?php
             } else echo " $i";
         }
 ?>
</p>
</body>
</html>
```

Figure 14.18 **A fragmented PHP script.** The `if` and `for` statements work as one might expect, despite the intervening raw HTML. When requested by a browser, this page displays the numbers from 0 to 19, with odd numbers written in bold.

Web designers are free to use whichever approach (`echo` or escape to raw HTML) seems most convenient for the task at hand.

Self-Posting Forms

EXAMPLE 14.81

Adder **web form with a PHP script**

By changing the `action` attribute of the `FORM` element, we can arrange for the Adder page of Figure C-14.16 to invoke a PHP script instead of a CGI script:

```
<form action="add.php" method="post">
```

The PHP script itself is shown in the top half of Figure C-14.19. Form values are made available to the script in an associative array (hash table) named `_REQUEST`. No special library is required.

EXAMPLE 14.82

Self-posting Adder **web form**

Because our PHP script is executed directly by the web server, it can safely reside in an arbitrary web directory, including the one in which the Adder page resides. In fact, by checking to see how a page was requested, we can merge the form and the script into a single page, and let it service its own requests! We illustrate this option in the bottom half of Figure C-14.19.

14.3.3 Client-Side Scripts

While embedded server-side scripts are generally faster than CGI scripts, at least when start-up cost predominates, communication across the Internet is still too slow for truly interactive pages. If we want the behavior or appearance of the page

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Adder</title></head>
<body><p>
<?php
      $argA = $_REQUEST['argA']; $argB = $_REQUEST['argB'];
      $sum = $argA + $argB;
      echo "$argA plus $argB is $sum\n";
 ?>
</p></body></html>
```

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8">
<?php
    $argA = $_REQUEST['argA'];  $argB = $_REQUEST['argB'];
    if ($argA == "" || $argB == "") {
?>
        <title>Adder</title></head><body>
        <form action="adder.php" method="post">
        <p><input name="argA" size="3"> First addend<br>
           <input name="argB" size="3"> Second addend</p>
        <p><input type="submit"></p>
        </form></body></html>
<?php
    } else {
?>
        <title>Sum</title></head><body><p>
<?php
        $sum = $argA + $argB;
        echo "$argA plus $argB is $sum\n";
?>
        </p></body></html>
<?php
    }
?>
```

Figure 14.19 **An interactive PHP web page.** The script at top could be used in place of the script in the middle of Figure C-14.16. The lower script in the current figure replaces both the web page at the top and the script in the middle of Figure C-14.16. It checks to see if it has received a full set of arguments. If it hasn't, it displays the fill-in form; if it has, it displays results.

to change as the user moves the mouse, clicks, types, or hides or exposes windows, we really need to execute some sort of script on the client's machine.

Because they run on the web designer's site, CGI scripts and, to a lesser extent, embeddable server-side scripts can be written in many different languages. All the client ever sees is standard HTML. Client-side scripts, by contrast, require an interpreter on the client's machine. By virtue of having been "in the right place at the right time" historically, JavaScript is supported with at least some degree of consistency by almost all of the world's web browsers. Moreover, given the number of legacy browsers still running, and the difficulty of convincing users to upgrade or to install new plug-ins, it has been difficult for any other option for client-side scripting to gain traction. Only recently, with the advent of WebAssembly, has the dominance of JavaAcript begun to wane.

EXAMPLE 14.83
Adder web form in JavaScript

Figure C-14.20 shows a page with embedded JavaScript that imitates (on the client) the behavior of the Adder scripts of Figures C-14.16 and C-14.19. Function `doAdd` is defined in the header of the page so it is available throughout. In particular, it will be invoked when the user clicks on the Calculate button. By default, the input values are character strings; we use the `parseInt` function to convert them to integers. The parentheses around `(argA + argB)` in the final assignment statement then force the use of integer addition. The other occurrences of `+` are string concatenation. To disable the usual mechanism whereby input data are submitted to the server when the user hits the enter or return key, we have specified a dummy behavior for the `onsubmit` attribute of the form.

Rather than replace the page with output text, as our CGI and PHP scripts did, we have chosen in our JavaScript version to append the output at the bottom. The HTML `SPAN` element provides a named place in the document where this output can be inserted, and the `getElementById` JavaScript method provides us with a reference to this element. The HTML *Document Object Model (DOM)*, standardized by the World Wide Web Consortium (W3C), specifies a very large number of other elements, attributes, and user actions, all of which are accessible in JavaScript. Through them scripts can, at appropriate times, inspect or alter almost any aspect of the content, structure, or style of a page. ■

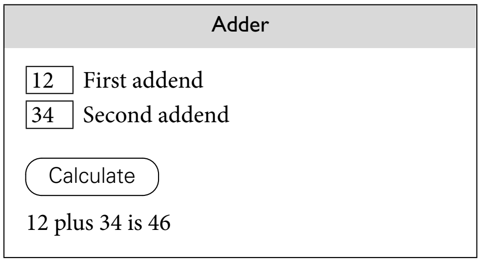
14.3.4 Java Applets and Other Embedded Elements

As an alternative to requiring client-side scripts to interact with the DOM of a web page, many browsers once supported an *embedding* mechanism that allowed a browser plug-in to assume responsibility for some rectangular region of the page, in which it could then display whatever it wanted. In other words, plug-ins were less a matter of scripting the browser than of bypassing it entirely. Historically, they were widely used for content—animations and video in particular—that were poorly supported by early versions of HTML.

EXAMPLE 14.84
Embedding an applet in a web page

Programs designed to be run by a Java plug-in were commonly known as *applets*. Consider, for example, an applet to display a clock with moving hands. Legacy browsers supported several different applet tags, but as of HTML5 the standard syntax looked like this:

```
<!DOCTYPE html>
<html lang="en">
<head><meta charset="utf-8"><title>Adder</title>
<script type="text/javascript">
function doAdd() {
    argA = parseInt(document.adder.argA.value)
    argB = parseInt(document.adder.argB.value)
    x = document.getElementById('sum')
    while (x.hasChildNodes())
        x.removeChild(x.lastChild)  // delete old content
    t = document.createTextNode(argA + " plus "
        + argB + " is " + (argA + argB))
    x.appendChild(t)
}
</script>
</head>
<body>
<form name="adder" onsubmit="return false">
<p><INPUT name="argA" size=3> First addend<br>
   <INPUT name="argB" size=3> Second addend</p>
<p><input type="button" onclick="doAdd()" value="Calculate"></p>
</form>
<p><span id="sum"></span></p>
</body>
</html>
```


Adder

12 First addend
34 Second addend

12 plus 34 is 46

Figure 14.20 **An interactive JavaScript web page.** Source appears at left. The rendered version on the right shows the appearance of the page after the user has entered two values and hit the Calculate button, causing the output message to appear. By entering new values and clicking again, the user can calculate as many sums as desired. Each new calculation will replace the output message.

```
<embed type="application/x-java-applet" code="Clock.class">
```

The `type` attribute informed the browser that the embedded element was expected to be a Java applet; the `code` element provided the applet's URI. Additional attributes could be used to specify such properties as the required interpreter version number and the size of the needed display space. ■

As one might infer from the existence of the `type` attribute, `embed` tags (and similar `object` tags) can request execution by a variety of plug-ins—not just a Java Virtual Machine. Historically, the most widely used plug-in was Adobe's Flash Player. Though scriptable, Flash Player is more accurately described as a multimedia display engine than a general purpose programming language interpreter.

Over time, plug-ins have proven to be a major source of browser security bugs. Almost any nontrivial plug-in requires access to operating system services—network IO, local file space, graphics acceleration, and so on. Providing just enough service to make the plug-in useful—but not enough to allow it to do any harm—has proven extremely difficult. To address this problem, extensive multimedia support has been built into HTML5, allowing the browser itself to assume responsibility for much of what was once accomplished with plug-ins. Security is still a problem, but the number of software modules that must be trusted—and the number of points at which an attacker might try to gain entrance—is significantly reduced. Almost all browsers now disable Java by default. Most disable Flash as well.

✓ **CHECK YOUR UNDERSTANDING**

47. Explain the distinction between *server-side* and *client-side* web scripting.
48. List the tradeoffs between CGI scripts and embedded PHP.
49. Why are CGI scripts usually installed only in a special directory?
50. Explain how a PHP page can service its own requests.
51. Why might we prefer to execute a web script on the server rather than the client? Why might we sometimes prefer the client instead?
52. What is the HTML *Document Object Model*? What is its significance for client-side scripting?
53. What is the relationship between JavaScript and Java?
54. What is an *applet*? Why are applets usually not considered an example of scripting?
55. Why are Java applets and Flash objects no longer commonly supported by web browsers?

DESIGN & IMPLEMENTATION

14.12 JavaScript and Java

Despite its name, JavaScript has no connection to Java beyond some superficial syntactic similarity. The language was originally developed by Brendan Eich at Netscape Corp. in 1995. Eich called his creation *LiveScript*, but the company chose to rename it as part of a joint marketing agreement with Sun Microsystems, prior to its public release. Trademark on the JavaScript name is actually owned by Oracle, which acquired Sun in 2010.

Netscape's browser was the market leader in 1995, and JavaScript usage grew extremely fast. To remain competitive, developers at Microsoft added JavaScript support to Internet Explorer, but they used the name *JScript* instead, and they introduced a number of incompatibilities with the Netscape version of the language. A common version was standardized as *ECMAScript* by the European standards body in 1997 (and subsequently by the ISO), but major incompatibilities remained in the Document Object Models provided by different browsers. These have been gradually resolved through a series of standards from the W3C and WHATWG, but legacy pages and legacy browsers continue to plague web developers.

14.3.5 XSLT

Most readers will undoubtedly have had the opportunity to write—or at least to read—the HTML (hypertext markup language) used to compose web pages. HTML has, for the most part, a nested structure in which fragments of documents (*elements*) are delimited by *tags* that indicate their purpose or appearance. We saw in Section 14.2.2, for example, that top-level headings are delimited with `<h1>` and `</h1>`. HTML was inspired by an older standard known as SGML (standard generalized markup language), developed in the 1980s and used, among other things, to computerize both the Oxford English Dictionary and the technical documentation of Boeing Corp.

DESIGN & IMPLEMENTATION

14.13 How far can you trust a script?

Security becomes an issue whenever code is executed using someone else's resources. On a hosting machine, web servers are usually installed with very limited access rights, and with only a limited view of the host's file system. This strategy limits the set of pages accessible through the server to a well-defined subset of what would be visible to users logged into the hosting machine directly. By contrast, CGI scripts are separate executable programs, and can potentially run with the privileges of whoever installs them. To prevent users on the hosting machine from accidentally or intentionally passing their privileges to arbitrary users on the Internet, most system administrators configure their machines so that CGI scripts must reside in a special directory, and be installed by a trusted user. Embedded server-side scripts can reside in any file because they are guaranteed to run with the (limited) rights of the server itself.

A larger risk is posed by code downloaded over the Internet and executed on a client machine. Because such code is in general untrusted, it must be executed in a carefully controlled environment, sometimes called a *sandbox* (a place where a child can safely play), to prevent it from doing any damage. As a general rule, embedded JavaScript cannot access the local file system, memory management system, or network, nor can it manipulate documents from other sites. Java applets, likewise, have only limited ability to access external resources. Reality is a bit more complicated, of course: Sometimes a script needs access to, say, a temporary file of limited size, or a network connection to a trusted server. Mechanisms exist to certify sites as *trusted*, or to allow a trusted site to certify the trustworthiness of pages from other sites. Scripts on pages obtained through a trusted mechanism may then be given extended rights. Such mechanisms must be used with care. Finding the right balance between security and functionality remains one of the central challenges of the Web, and of distributed computing in general. (More on this topic can be found in Sections 15.2.3 and 16.2.4, and in Explorations 16.21 and 16.22.)

In the early days of the Web, SGML was clearly too complex and formal for web pages, which needed to be written by hand and rendered in real time by slow computers. The simpler HTML evolved in an informal and ad hoc way, with incompatible extensions made by competing vendors. Standardization has been a long and difficult process: incompatibilities among browsers continue to frustrate web designers, and several features of the language that have been deprecated[3] in the most recent standards are nonetheless still widely used. Other features, while not deprecated, are widely regarded in hindsight to have been mistakes.

EXAMPLE 14.85

Content versus presentation in HTML

Probably the biggest problem with HTML is that it does not adequately distinguish between the *content* and the *presentation* (appearance) of a document. As a trivial example, web designers sometimes use `<i>...</i>` tags to request that text be set in an italic font, when `<em>...</em>` (emphasis) would be more appropriate. A browser for the visually impaired might choose to emphasize text with something other than italics, and might render book titles (also often specified with `<i>...</i>`) in some entirely different fashion. More significantly, many web designers use tables (`<table>...</table>`) to control the relative positioning of elements on a page, when the content isn't tabular at all. As the Web has extended across cell phones, televisions, tablets, watches, and audio-only devices, the need to distinguish between content and presentation has become increasingly essential. ■

This is where XML stepped in. A streamlined descendant of SGML, developed by the W3C in the mid to late 1990s, XML has at least three important advantages over HTML for data and document representation: (1) its syntax and semantics are more regular and consistent, and more consistently implemented across platforms; (2) it is *extensible*, meaning that users can define new tags; (3) it specifies content only, leaving presentation to a companion standard known as XSL (extensible stylesheet language). XSLT is a portion of XSL devoted to *transforming* XML: selecting, reorganizing, and modifying tags and the elements they delimit—in effect, scripting the processing of data represented in XML.

Internet Alphabet Soup

Learning about web standards can be a daunting task: there is an enormous number of buzzwords, standards, and multiletter abbreviations. The standards—and the relationships among them—are also moving targets, promulgated by groups whose interests are not always in sync. To start, it may help to note that each of the major markup languages—SGML, HTML, and XML—has a corresponding *stylesheet language*: DSSSL, CSS, and XSL, respectively. A stylesheet language is used to control the presentation of a document, separate from its content. Stylesheet languages are essential for SGML and XML; without them there is no way to know whether a `<RECORD>` represents a database entry, an antique phonograph album, or an Olympic achievement, much less how to display it. HTML is less dependent

[3] A *deprecated* feature is one whose use is officially discouraged, but permitted on a temporary basis, to ease the transition to new and presumably better alternatives.

on stylesheets, but most professionally maintained web sites use CSS to create a uniform “look and feel” across a collection of pages without embedding redundant information in every page.

SGML is still used for large-scale projects in the business world, though many newer projects have chosen to use XML or JSON, the JavaScript Object Notation. JSON is more compact and self-descriptive than XML, and is commonly used to transmit structured data between web servers and clients. It does not have a stylesheet language comparable to XSL, however. HTML continues to evolve (see sidebar C-14.14). HTML5, codified by the World Wide Web Consortium in 2014, added extensive support for multimedia content, and specified both general and XML-compliant versions of the syntax.

XML and XHTML

As a general rule, the syntax of XML is simpler than that of SGML or HTML. To allow XML tools (XSLT in particular) to be used to process web pages, the HTML5 standard defines a restricted version of the HTML syntax, known as XHTML. With a few minor exceptions, any web page that can be specified in HTML can also be specified in XHTML, and vice versa. The `content-type` header that precedes a web page when transmitted over the Internet tells the browser which parser to use:

DESIGN & IMPLEMENTATION

14.14 W3C and WHATWG

Standardization efforts for HTML have a complicated history. With the completion in 1998 of the XML 1.0 specification, the World Wide Web Consortium (W3C) focused on XHTML, in an effort to push the world toward a “cleaned-up,” XML-compliant version of HTML. Over the next few years, this strategy proved increasingly contentious. In 2004, a group of influential individuals from Apple, Mozilla, and Opera split off to form a separate Web Hypertext Application Technology Working Group (WHATWG), with the goal of evolving HTML in a way that preserved complete backward compatibility and interoperability. In 2006, the W3C reconsidered its position, and began to work with WHATWG toward what eventually became HTML5. Both groups continued to work on HTML evolution, largely but not entirely in sync. In 2019, they agreed that future development would belong to WHATWG, though W3C continues to participate.

While W3C would prefer a dated, finalized document (which it would number HTML5), WHATWG’s “Living Standard” for HTML has been continuously evolving (without version numbers) since 2012. WHATWG believes that the standard should reflect without necessarily dictating current practice, as embodied in the browsers of all major vendors. Both the W3C and WHATWG distinguish carefully between what a conforming document should contain and what a conforming browser should be able to render: the latter is significant superset of the former.

`text/html` means "regular" HTML; `application/xhtml+xml` means XHTML. In practice, the principal differences between the notations are that XHTML is harder for human beings to write, because the rules are stricter, and XML parsers are designed to reject (and decline to render) any page that is not *well formed* (syntactically correct). HTML parsers are designed to tolerate—and do something reasonable with—even the worst "tag soup." With some care, it is possible to write pages that will be processed correctly by both HTML and XHTML parsers; such pages are said to use *polyglot markup* (syntax).

In any well-formed XML document (including those written in XHTML), tags must either constitute properly nested, matched pairs, or be explicit singletons, which end with a "`/>`" delimiter. Similarly, the values of *attributes* (key–value pairs embedded within tags) must always be specified with quotes. The following fragment, for example, is well formed (though incomplete) XHTML:

EXAMPLE 14.86
Well-formed XHTML

```
<em><q id="favorite">I defy the tyranny of precedent</q></em><br />
(Clara Barton)
```

Here the quotation element (`<q>` ... `</q>`) is nested inside the emphasis element (`<em>` ... `</em>`). Moreover the "break" element (`<br />`), which usually causes subsequent text to start on a new line, is explicitly a singleton; it has a slash before its closing "`>`" delimiter. (To avoid confusing certain legacy browsers, one sometimes needs a space in front of the slash.) The example fragment would be malformed if the slash were missing, if the opening `<em><q>` tags were reversed (`<q><em>`), or if the attribute value `"favorite"` had not be enclosed in quote marks. An HTML parser would tolerate these errors; an XML parser will not.

The set of tags to be used in an XML document can be specified by naming a *document type definition* (DTD) in the document's `DOCTYPE` header, or by naming an *XML Schema* in an attribute of the document's top-level tag. (XML Schemas are a newer format, but DTDs remain in widespread use.) Among other things, a DTD or Schema indicates which tags are allowed, whether those tags are pairs or singletons, whether they permit attributes, and whether any attributes are mandatory. If a document has no DTD or Schema, it is said to define a DTD *implicitly* by virtue of which tags are actually used. Implicit definition suffices for the examples in this chapter.

EXAMPLE 14.87
XHTML to display a favorite quote

Because tags must nest in XML, a document has a natural tree-based structure. Figure C-14.21 shows the source for a small but complete polyglot HTML5 document, together with the tree it represents. There are three kinds of nodes in the tree: elements (delimited by tags in the source), text, and attributes. The internal (nonleaf) nodes are all elements. Everything nested between the beginning and ending tags of an element is an attribute or child of that element in the tree.

The root of our document, named "`/`" by convention, has one child—the `html` element. This in turn has three attributes—`xmlns`, `lang`, and `xml:lang`—and two child elements—`head` and `body`. The `xmlns` attribute specifies a URI for our document's *namespace*. This serves a purpose similar to that of C++ namespaces or Java packages (Section 3.8): it allows us to give tag names a disambiguating

```
<!DOCTYPE html>
<html xmlns="https://www.w3.org/1999/xhtml" lang="en" xml:lang="en">
<head>
<meta charset="UTF-8" />
<title>Favorite Quote</title>
</head>
<body>
<p>
<em><q id="favorite">I defy
the tyranny of precedent</q></em><br />
(Clara Barton)
</p>
</body>
</html>
```

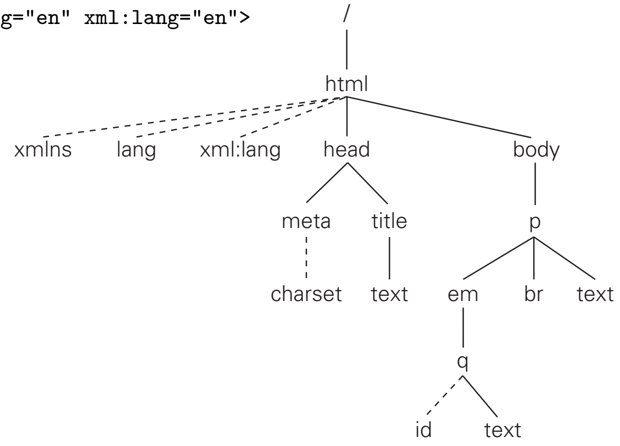


Figure 14.21 **A complete XHTML document and its corresponding tree.** Child elements are shown with solid lines, attributes with dashed lines.

prefix: `xhtml:table` versus `furniture:table`. With the value we have specified for the `xmlns` attribute, any tag in the document that doesn't have a prefix will automatically be interpreted as being in the `xhtml` namespace. The `lang` and `xml:lang` tags specify the source language (English) for HTML and XML parsers, respectively.

XSLT and XPath

XSL (extensible stylesheet language) can be thought of as a language for specifying what to *do* with an XML document. It has four sublanguages, called XSLT, XPath, XSL-FO, and XQuery. XSLT is a scripting language that takes XML as input and produces textual output—often transformed XML or HTML, but potentially other formats as well.

EXAMPLE 14.88

XPath names for XHTML elements

XPath is a language used to name things in XML documents. XPath names frequently appear in the attributes of XSLT elements. Returning to Figure C-14.21, the quotation element of our document could be named in XPath as `/html/body/p/em/q`. The emphasis element and its break and text-node siblings, together, could be named as `/html/body/p/*`. XPath includes a rich set of naming mechanisms, including absolute (from the root) and relative (from the current node) navigation, wildcards, predicates, substring and regular expression manipulation, and counting and arithmetic functions. We will see some of these in the extended example below.

XSL-FO (XSL formatting objects) is a set of tags to specify the *layout* (presentation) of a document, in terms of pages, regions (e.g., header, body, footer), blocks (paragraph, table, list), lines, and in-line elements (character, image). An XSLT script might be used to add XSL-FO tags to an XML document, or to transform a document that already has XSL-FO tags in it—perhaps to split a long single-page

```xml
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="bib.xsl"?>
<bibliography>
  <book>
    <author>Guido van Rossum</author>
    <editor>Fred L. Drake, Jr.</editor>
    <title>The Python Language Reference Manual (version 3.2)</title>
    <publisher>Network Theory, Ltd.</publisher>
    <address>Bristol, UK</address>
    <year>2011</year>
    <note>Available at <uri>https://books.google.com/books/about
        /The_Python_Language_Reference_Manual.html?id=Ut4BuQAACAAJ</uri></note>
  </book>
  <article>
    <author>John K. Ousterhout</author>
    <title>Scripting: Higher-Level Programming for the 21st Century</title>
    <journal>Computer</journal>
    <volume>31</volume>
    <number>3</number>
    <month>March</month>
    <year>1998</year>
    <pages>23&#8211;30</pages>
  </article>
  <inproceedings>
    <author>Theodor Holm Nelson</author>
    <title>Complex Information Processing: A File Structure for the
        Complex, the Changing, and the Indeterminate</title>
    <booktitle>Proceedings of the Twentieth ACM National Conference</booktitle>
    <month>August</month>
    <year>1965</year>
    <address>Cleveland, OH</address>
    <pages>84&#8211;100</pages>
  </inproceedings>
  <inproceedings>
    <author>Stephan Kepser</author>
    <title>A Simple Proof for the Turing-Completeness of XSLT and XQuery</title>
    <booktitle>Proceedings, Extreme Markup Languages 2004</booktitle>
    <address>Montr&#233;al, Canada</address>
    <year>2004</year>
    <month>August</month>
    <note>Available at <uri>https://citeseerx.ist.psu.edu/document?
        doi=5f7ad1d9c17c01e3321b44ad996ff3fcd3ddbea3</uri></note>
  </inproceedings>
```

Figure 14.22 **A bibliography in XML.** References (two books, a journal article, and three conference papers) appear in arbitrary order. The two URIs have been wrapped to fit on the printed page. *(continued)*

```
  <inproceedings>
    <author>David G. Korn</author>
    <title><code>ksh</code>: An Extensible High Level Language</title>
    <booktitle>Proceedings of the USENIX Very High Level Languages Symposium</booktitle>
    <address>Santa Fe, NM</address>
    <year>1994</year>
    <month>October</month>
    <pages>129&#8211;146</pages>
  </inproceedings>
  <book>
    <author>Tom Christiansen</author>
    <author>brian d foy</author>
    <author>Larry Wall</author>
    <author>Jon Orwant</author>
    <title>Programming Perl</title>
    <edition>fourth</edition>
    <publisher>O&#8217;Reilly Media</publisher>
    <address>Sebastopol, CA</address>
    <year>2012</year>
  </book>
</bibliography>
```

Figure 14.22 *(continued)*

document intended for the Web into a multipage document intended for printing on paper.

XQuery is a language in which to frame information-retrieval questions for a database stored in XML format. (In a bibliographic database, for example, we might use XQuery look for journal articles written since the turn of the century.) The purpose and behavior of XQuery parallel those of SQL, the standard language used for relational database queries. For the sake of simplicity, we will not use XSL-FO or XQuery in our extended example. Rather we will peruse an entire XML document, using XSLT to format its content as HTML.

An XML document can explicitly specify an XSLT script that should be used to transform or format it. This is a common but somewhat restrictive way to go about things: by tying a single stylesheet to the XML file we compromise the separation between content and presentation that was a principal motivation for creating XML in the first place. An alternative is to use client-side JavaScript or server-side PHP to invoke the XSLT processor, passing the XML document and the XSLT script as arguments. As of 2023, the XSLT 3 is the newest version of the language. XSLT 1 support is included in all major browsers; newer versions typically require a JavaScript library.

Extended Example: Bibliographic Formatting

EXAMPLE 14.89

Creating a reference list with XSLT

As an example of a task for which we might realistically use XSLT, consider the creation of a bibliographic reference list. Figure C-14.22 contains XML source for

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="https://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <html><head><title>Bibliography</title></head><body><h1>Bibliography</h1><ol>
    <xsl:for-each select="bibliography/*"><xsl:sort select="title"/>
      <li><xsl:apply-templates select="."/></li>
    </xsl:for-each>
  </ol></body></html>
</xsl:template>

<xsl:template match="bibliography/article">
  <q><xsl:apply-templates select="title/node()"/>,</q>
  by <xsl:call-template name="author-list"/>.&#160;
  <em><xsl:apply-templates select="journal/node()"/>
  <xsl:text> </xsl:text><xsl:apply-templates select="volume/node()"/>
  </em>:<xsl:apply-templates select="number/node()"/>
  (<xsl:apply-templates select="month/node()"/><xsl:text> </xsl:text>
    <xsl:apply-templates select="year/node()"/>),
  pages <xsl:apply-templates select="pages/node()"/>.
  <xsl:if test="note"><xsl:apply-templates select="note/node()"/>.</xsl:if>
</xsl:template>

<xsl:template match="bibliography/book">
  <em><xsl:apply-templates select="title/node()"/>,</em>
  by <xsl:call-template name="author-list"/>.&#160;
  <xsl:apply-templates select="publisher/node()"/>,
  <xsl:apply-templates select="address/node()"/>,
  <xsl:if test="edition">
    <xsl:apply-templates select="edition/node()"/> edition, </xsl:if>
  <xsl:apply-templates select="year/node()"/>.
  <xsl:if test="note"><xsl:apply-templates select="note/node()"/>.</xsl:if>
</xsl:template>

<xsl:template match="bibliography/inproceedings">
  <q><xsl:apply-templates select="title/node()"/>,</q>
  by <xsl:call-template name="author-list"/>.&#160;
  In <em><xsl:apply-templates select="booktitle/node()"/></em>
  <xsl:if test="pages">, pages <xsl:apply-templates select="pages/node()"/></xsl:if>
  <xsl:if test="address">, <xsl:apply-templates select="address/node()"/></xsl:if>
  <xsl:if test="month">, <xsl:apply-templates select="month/node()"/></xsl:if>
  <xsl:if test="year">, <xsl:apply-templates select="year/node()"/></xsl:if>.
  <xsl:if test="note"><xsl:apply-templates select="note/node()"/>.</xsl:if>
</xsl:template>
```

Figure 14.23 Bibliography stylesheet in XSL. This script will generate HTML when applied to a bibliography like that of Figure C-14.22. *(continued)*

```
<xsl:template name="author-list">       <!-- format author list -->
  <xsl:for-each select="author|editor">
    <xsl:if test="last() > 1 and position() = last()"> and </xsl:if>
    <xsl:apply-templates select="./node()"/>
    <xsl:if test="self::editor"> (editor)</xsl:if>
    <xsl:if test="last() > 2 and last() > position()">, </xsl:if>
  </xsl:for-each>
</xsl:template>

<xsl:template match="uri">              <!-- format link -->
    <a><xsl:attribute name="href"><xsl:value-of select="."/></xsl:attribute>
    <xsl:value-of select="substring-after(., 'https://')"/></a>
</xsl:template>

<xsl:template match="@*|node()">        <!-- default: copy content -->
  <xsl:copy><xsl:apply-templates select="@*|node()"/></xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

Figure 14.23 *(continued)*

such a list. (Field names have been borrowed from BibTeX [Lam94, App. B].) The document begins with a declaration to specify the XML version and character encoding, and a processing instruction to specify the XSL stylesheet to be used to format the file. These declarations are included for the benefit of tools that process the document; they aren't part of the XML source itself. (Note the syntactic resemblance to the *processing instructions* used in Section C-14.3.2 to provide input to the PHP interpreter.)

At the top level, the `bibliography` element consists of a series of `book`, `article`, and `inproceedings` elements, each of which may contain elements for author and editor names, title, publisher, date and address, and so on. Some elements may contain nested `uri` elements, which specify on-line links. Characters that cannot be represented in ASCII are shown as Unicode *character entities*, as described in Sidebar 7.3.

Figure C-14.23 contains an XSLT stylesheet (script) to format the bibliography as HTML, which may then be rendered in a browser. This script was named at the beginning of the XML document (Figure C-14.22). In a manner analogous to that of the XML document, the script begins with a declaration to specify the XML version and character encoding, and an `xsl:stylesheet` element to specify the XSL version and namespace. The remainder of the script contains a mix of XSL and HTML elements. The XSL tags all specify the `xsl:` namespace explicitly. They are recognized by the XSLT processor. Elements from other namespaces are treated as ordinary text, to be copied through to the output when encountered.

The fundamental construct in XSLT is the `template`, which specifies a set of *instructions* to be applied to nodes in an XML source tree. Templates are typically invoked by executing an apply-templates or a call-template instruction in some other template. Each invocation has a concept of *current node*. The execution as a whole begins by invoking an initial template with the root of the source tree (/) as current node. In our bibliographic example, the initial template is the one at the top of the script, because its match attribute is the XPath expression "/". The body of the initial template begins with a string of HTML elements and text. This string is copied directly to the output. The for-each element, however, is an XSLT instruction, so it is executed.

The select attribute of the for-each element uses an XPath expression ("bibliography/*") to build a *node set* consisting of all top-level entries in our bibliography. Other expressions could have been used if we wanted to be selective: "bibliography/*[year>=2000]" would match only recent entries; "bibliography/*[note]" would match only entries with note elements; "bibliography/article|bibliography/book" would match only articles and books.

The nested sort instruction forces the selected node set to be ordered alphabetically by title. The body of the for-each is then executed with each entry in turn selected as current node. The body contains a recursive invocation of apply-templates, bracketed by HTML list tags (<li> … </li>). These tags are copied to the output, with the result of the recursive call nested in between.

So how does the recursive call work? Its select attribute, like that of for-each, uses XPath to build a node set. In this case it is the trivial node set containing only ".", the current node of the current iteration of for-each. The XSLT processor searches for a template that matches this node. We have created three appropriate candidates, one for each kind of bibliographic entry. When it finds the matching template, the processor invokes it, with an updated notion of current node.

Each of our three main templates contains a set of instructions to format its kind of entry (article, book, conference paper). Most of the instructions use additional invocations of apply-templates to format individual portions of an entry (author, title, publisher, etc.). Interspersed in these instructions are snippets of text and HTML elements. In several cases we use an if instruction to generate output only when a given XML element is present in the source. In most of these the recursive call uses the XPath node() function to select all children of the element in question.

White space is ignored when it comes between the end of one instruction and the beginning of the next. To force white space into the output in this case, we must delimit it with <text> … </text> tags. Extra white space (e.g., after the ends of sentences) is specified with the "nonbreaking space" character entity, &#160;.

Three extra templates end our script. The most interesting of these serves to format author lists. It has a name attribute rather than a match attribute, and is invoked with call-template rather than apply-templates. A called template always takes the current node of the caller—in this case the node that represents a bibliographic entry. Internally, the author list template executes a for-each instruction that selects all child nodes representing authors or editors. The for-each, in turn, uses the XPath last() and position() functions to determine how many

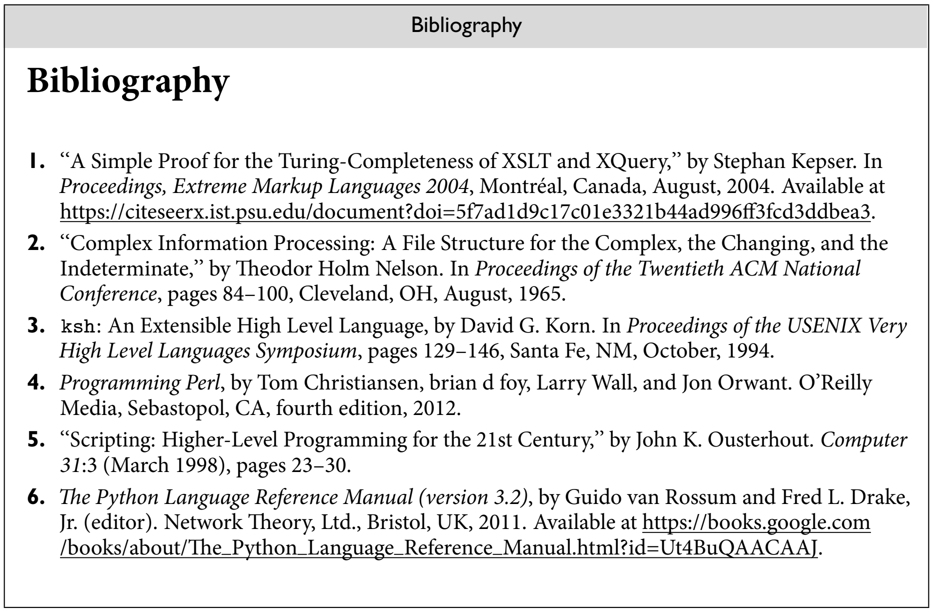
```
<html><head><title>Bibliography</title></head>
<body><h1>Bibliography</h1><ol>
<li>
  <q>A Simple Proof for the Turing-Completeness of XSLT and XQuery,</q>
  by Stephan Kepser.&nbsp; In <em>Proceedings, Extreme Markup Languages
  2004</em>, Montr&eacute;al, Canada, August, 2004.  Available at
  <a href="https://citeseerx.ist.psu.edu/document?doi=
5f7ad1d9c17c01e3321b44ad996ff3fcd3ddbea3">citeseerx.ist.psu.edu
/document?doi=5f7ad1d9c17c01e3321b44ad996ff3fcd3ddbea3</a>.</li>
<li>
  <q>Complex Information Processing: A File Structure for the Complex,
  the Changing, and the Indeterminate,</q> by Theodor Holm Nelson.&nbsp;
  In <em>Proceedings of the Twentieth ACM National Conference</em>,
  pages 84&ndash;100, Cleveland, OH, August, 1965.</li>
<li>
  <q><code>ksh</code>: An Extensible High Level Language,</q> by David
  G. Korn.&nbsp; In <em>Proceedings of the USENIX Very High Level Languages
  Symposium</em>, pages 129&ndash;146, Santa Fe, NM, October, 1994.</li>
<li>
  <em>Programming Perl,</em> by Tom Christiansen, brian d foy, Larry Wall,
  and Jon Orwant.&nbsp; O&rsquo;Reilly Media, Sebastopol, CA, fourth
  edition, 2012.</li>
<li>
  <q>Scripting: Higher-Level Programming for the 21st Century,</q> by
  John K. Ousterhout.&nbsp; <em>Computer 31</em>:3 (March 1998), pages
  23&ndash;30.</li>
<li>
  <em>The Python Language Reference Manual (version 3.2),</em> by Guido
  van Rossum and Fred L. Drake, Jr. (editor).&nbsp; Network Theory, Ltd.,
  Bristol, UK, 2011.  Available at <a href="https://books.google.com/books/about
/The_Python_Language_Reference_Manual.html?id=Ut4BuQAACAAJ">books.google.com/books
/about/The_Python_Language_Reference_Manual.html?id=Ut4BuQAACAAJ</a>.</li>
</ol>
</body></html>
```

Figure 14.24 Result of applying the stylesheet of Figure C-14.23 to the bibliography of Figure C-14.22.

names there are, and where each name falls in the list. It inserts the word "and" between the final two names, and puts commas after all names but the last in lists of three or more.

The template with `match="uri"` serves to format URIs that appear anywhere in the XML source. It creates an HTML link in the output, but uses the XPath `substring-after` function to strip the leading *https://* off the visible text. XPath provides a variety of similar functions for string and regular expression manipulation. The `value-of` instruction copies the contents of the selected node to the output, as a character string.

Our final template serves as a default case. The XPath expression `"@*|node()"` will match any attribute or other node in the XML source. Inside, the `copy` instruc-



Bibliography

Bibliography

1. “A Simple Proof for the Turing-Completeness of XSLT and XQuery,” by Stephan Kepser. In *Proceedings, Extreme Markup Languages 2004*, Montréal, Canada, August, 2004. Available at https://citeseerx.ist.psu.edu/document?doi=5f7ad1d9c17c01e3321b44ad996ff3fcd3ddbea3.
2. “Complex Information Processing: A File Structure for the Complex, the Changing, and the Indeterminate,” by Theodor Holm Nelson. In *Proceedings of the Twentieth ACM National Conference*, pages 84–100, Cleveland, OH, August, 1965.
3. `ksh`: An Extensible High Level Language, by David G. Korn. In *Proceedings of the USENIX Very High Level Languages Symposium*, pages 129–146, Santa Fe, NM, October, 1994.
4. *Programming Perl*, by Tom Christiansen, brian d foy, Larry Wall, and Jon Orwant. O’Reilly Media, Sebastopol, CA, fourth edition, 2012.
5. “Scripting: Higher-Level Programming for the 21st Century,” by John K. Ousterhout. *Computer 31*:3 (March 1998), pages 23–30.
6. *The Python Language Reference Manual (version 3.2)*, by Guido van Rossum and Fred L. Drake, Jr. (editor). Network Theory, Ltd., Bristol, UK, 2011. Available at https://books.google.com/books/about/The_Python_Language_Reference_Manual.html?id=Ut4BuQAACAAJ.

Figure 14.25 Rendered version of the HTML in Figure C-14.24.

tion copies the node’s tags, if any, to the output, with the result of a recursive call to `apply-templates` in between. The `"@*|node()"` on the recursive call selects a node set consisting of all the current node’s attributes and children. The end result is that any XML elements in the source that are delimited by tags for which we do not have special templates will be regenerated in the output just as they appear in the source. The recursion stops at text nodes and attributes, which are the leaves of the XML tree.

HTML output from our script appears in Figure C-14.24. The rendered web page appears in Figure C-14.25.

While lengthy by the standards of this text, our example illustrates only a fraction of the capabilities of XSLT. In the standard categorization of programming languages, the notation is strongly declarative: values may have names, but there are no mutable variables, and no side effects. There is a limited looping mechanism (`for-each`), but the real power comes from recursion, and from recursive traversal of XML trees in particular. ■

✓ **CHECK YOUR UNDERSTANDING**

56. Explain the relationships among SGML, HTML, and XML. What are their corresponding stylesheet languages?

57. Why does XML work so hard to distinguish between *content* and *presentation*?

58. What are the four main components of XSL? What are their respective purposes?
59. What is XHTML? How does it differ from “ordinary” HTML?
60. Explain the correspondence between XML documents and trees.
61. What does it mean for an XML document to be *well formed*?
62. Explain the distinctions (syntactic and semantic) among *elements*, *declarations*, and *processing instructions* in XML. Also explain the distinctions among *elements*, *tags*, and *attributes*.
63. Summarize the execution model of XSLT. In a nutshell, how does it work?
64. Explain the difference between *applying* templates and *calling* them in XSLT.

14 Scripting

14.6 Exercises

14.15 Explain the circumstances under which it makes sense to realize an interactive task on the Web as a CGI script, an embedded server-side script, or a client-side script. For each of these implementation choices, give three examples of tasks for which it is clearly the preferred approach.

14.16 (a) Write a web page with embedded PHP to print the first 10 rows of Pascal's triangle (see Example C-17.10 if you don't know what this is). When rendered, your output should look like Figure C-14.26.

(b) Modify your page to create a self-posting form that accepts the number of desired rows in an input field.

(c) Rewrite your page in JavaScript.

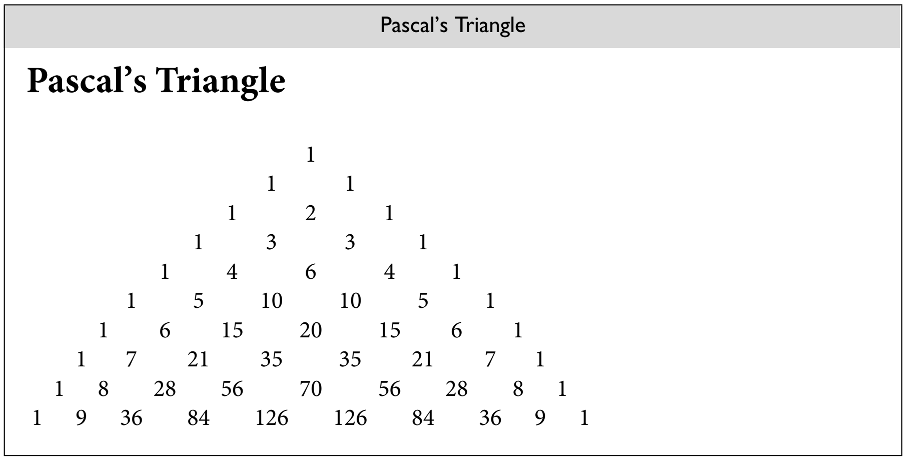
14.17 Create a fill-in web form that uses a JavaScript implementation of the Luhn formula (Exercise C-4.27) to check for typos in credit card numbers. (But don't use real credit card numbers; homework exercises don't tend to be very secure!)

14.18 (a) Modify the code of Figure C-14.20 (Example C-14.83) so that it replaces the form with its output, as the CGI and PHP versions of Figures C-14.16 and C-14.19 do.

(b) Modify the CGI and PHP scripts of Figures C-14.16 and C-14.19 (Examples C-14.78 and C-14.82) so they appear to append their output to the bottom of the form, as the JavaScript version of Figure C-14.20 does.

14.19 Modify the XSLT of Figure C-14.23 to do one or more of the following:

(a) Alter the titles of conference papers so that only first words, words that follow a dash or colon (and thus begin a subtitle), and proper nouns are capitalized. You will need to adopt a convention by which the creator of the document can identify proper nouns.



Pascal’s Triangle

Pascal’s Triangle

```
                           1
                        1     1
                     1     2     1
                  1     3     3     1
               1     4     6     4     1
            1     5    10    10     5     1
         1     6    15    20    15     6     1
      1     7    21    35    35    21     7     1
   1     8    28    56    70    56    28     8     1
1     9    36    84   126   126    84    36     9     1
```

Figure 14.26 Pascal’s triangle rendered in a web page (Exercise C-14.16).

(b) Sort entries by the last name of the first author or editor. You will need to adopt a convention by which the creator of the document can identify compound last names (“von Neumann,” for example, should be alphabetized under ‘v’).

(c) Allow bibliographic entries to contain an `abstract` element, which when formatted appears as an indented block of text in a smaller font.

(d) In addition to the `book`, `article`, and `inproceedings` elements, add support for other kinds of entries, such as manuals, technical reports, theses, newspaper articles, web sites, and so on. You may want to draw inspiration from the categories supported by BibTeX [Lam94, App. B].

(e) Format entries according to some standard style convention (e.g., that of the Chicago Manual of Style [*chicagomanualofstyle.org/book/ed17/part3/ch14/toc.html*] or the ACM Transactions [*acm.org/publications/authors/submissions*]).

14.20 Suppose bibliographic entries in Figure C-14.22 contain a mandatory `key` element, and that other documents can contain matching `cite` elements. Create an XSLT script that imitates the work of BibTeX. Your script should

(a) read an XML document, find all the `cite` elements, collect the keys they contain, and replace them with `bibref` elements that contain small integers instead.

(b) read a separate XML bibliography document, extract the entries with matching keys, and write them, in sorted order, to a new (and probably smaller) bibliography.

The small numbers in the bibref elements of the new document from (a) should match the corresponding numbered entries in the new bibliography from (b).

14.21 Write a program that will read an XHTML file and print an outline of its contents, by extracting all `<title>`, `<h1>`, `<h2>`, and `<h3>` elements, and printing them at varying levels of indentation. Write

(a) in C or Java

(b) in `sed` or `awk`

(c) in Perl, Python, or Ruby

(d) in XSLT

Compare and contrast your solutions.

14 Scripting

14.7 Explorations

14.32 Learn about Dart, a language developed at Google. Initially intended as a successor to JavaScript, Dart is now supported only as a language in which to develop code that will be *translated into* JavaScript. What explains the change in strategy?

14.33 Learn more about WebAssembly. Why has it been successful when previous proposed alternatives to JavaScript were not?

14.34 Learn more about DTDs and XML Schemas. Compare the DTD and XML Schema definitions of XHTML. What appear to the prospects for migrating to the newer specification language?

14.35 Academics often keep lists of publications in multiple places and formats: an on-line web page, a printable resume, a BIBTEX database for paper writing [Lam94, App. B]. Using XSLT, build a set of tools that will construct these lists automatically from a single XML source file.

14.36 Learn about XSL-FO. Use it to reimplement Example C-14.89. Your new version should be a two-stage process: one XSLT script should add formatting tags to the XML bibliography; a second should convert the tagged bibliography to XHTML. Try to make these stages as general as possible: you should be able to modify the appearance of the output list by changing the first script only. You should also be able to write alternative versions of the second script that generate output in formats other than XHTML (e.g., LaTeX).

14.37 Learn more about the history of W3C and WHATWG. What are the comparative advantages and disadvantages of their approaches to standardization? Do you find yourself more in sympathy with one approach or the other? How large are the technical differences between the most recent versions of the HTML standards? Are these differences significant enough to pose a problem for web developers?

15 Building a Runnable Program

15.2.1 GCC and LLVM

Traditionally, all machine-independent code improvement in `gcc` was based on RTL. Over time it became clear that the IF had become an obstacle to further improvements in the compiler, and that a higher-level form was needed. GIMPLE was introduced to meet that need. Since `gcc` v.4.9 (2014), GENERIC has been used for semantic analysis and, in a few cases, for certain language-specific code improvement. As its final task, each front end converts the program from GENERIC into GIMPLE. Depending on the requested level of code improvement, the "middle end" may perform over 140 phases of code improvement and transformation on the GIMPLE representation, after which it converts to RTL and performs as many as 70 additional phases before handing the result to the back end for target code generation.

Both GIMPLE and RTL are meant to be kept in memory across compiler phases, rather than being written to a file. Both IFs have a human-readable external format, which the compiler can write and (partially) read, but this format is not needed by the compiler: the internal version is much better suited for automatic manipulation.

GIMPLE

EXAMPLE 15.19
GCD program in GIMPLE

The GIMPLE code generated by a `gcc` front end is essentially a distillation of GENERIC, with many of the most complex (and often language-specific) features "lowered" into a smaller, common set of tree node types. As a simple example, consider the `gcd` program of Example 1.20:

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

Figure c-15.11 illustrates the "high GIMPLE" produced by `gcc`'s C front end when given this program as input. If we compare this GIMPLE code to Figure 15.2, which loosely[1] resembles GENERIC, we see at least two significant differences. First, all of the nodes that comprise a subroutine appear on a single list, with control flow represented by explicit `goto`s and by `true` and `false` branches for conditions. Second, both conditions and assignments have been designed to capture an embedded binary expression, allowing us in many cases to collapse a small subtree into a GIMPLE single node. ■

Over the course of its many phases, the `gcc` middle end will make many additional changes to this code, not only to improve its quality but also to further lower its level of abstraction. This "flattening" of the tree makes it easier to translate into RTL.

Perhaps the most significant transformation of GIMPLE is the conversion to *static single assignment (SSA) form*. As noted in the main text and explored more fully in Section c-17.4.1, SSA conversion facilitates subsequent code transformations by introducing extra variable names into the program in such a way that nothing is ever written in more than one place.

RTL

RTL is loosely based on the S-expressions of Lisp. Each RTL expression consists of an operator or expression type and a sequence of operands. In its external form, these are represented by a parenthesized list in which the element immediately inside the left parenthesis is the operator. Each such list is then embedded in a wrapper that points to predecessor and successor expressions in linear order. Internally, RTL expressions are represented by C structs and pointers. This pointer-rich structure constitutes the interface among the compiler's many back-end phases. There are several dozen expression types, including constants, references to values in memory or registers, arithmetic and logical operations, comparisons, bit-field manipulations, type conversions, and stores to memory or registers.

The body of a subroutine consists of a sequence of RTL expressions. Each expression in the sequence is called an `insn` (instruction). Each `insn` begins with one of six special codes:

insn: an "ordinary" RTL expression.

jump_insn: an expression that may transfer control to a label.

call_insn: an expression that may make a subroutine call.

code_label: a possible target of a jump.

barrier: an indication that the previous `insn` always jumps away. Control will never "fall through" to here.

[1] Unlike the informal notation of Figure 15.2, GENERIC and GIMPLE make no distinction between syntax tree nodes and symbol table nodes. In effect, the symbol table is merged into the syntax tree.

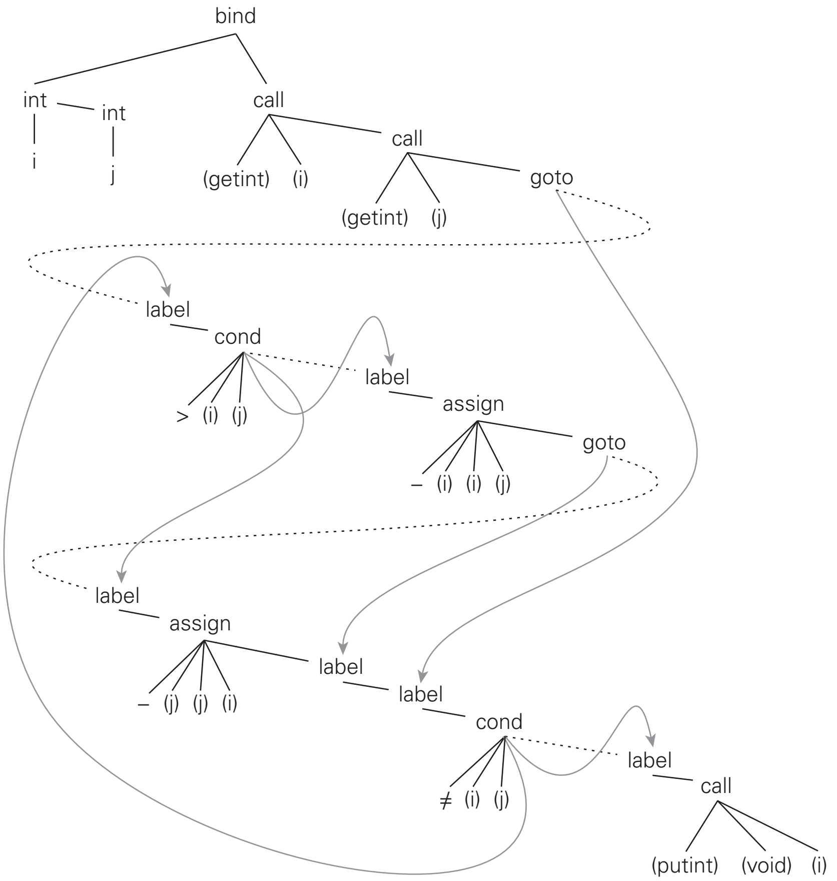


Figure 15.11 **Simplified GIMPLE for the `gcd` program.** The left child of the `bind` node holds local symbol table information; references to this information—and to global functions getint and putint—are indicated in the rest of the figure with parenthesized names. The first child of a call node names the function, the second the place to assign the return value, and the rest the arguments. An assign node with children ⟨op, a, b, c⟩ represents the assignment a := b op c. In each condition node, the first three children are a comparison operator and its operands; the last two are pointers to the subtrees for the outcomes true and false.

note: a pure annotation. There are nine different kinds of these, to identify the tops and bottoms of loops, scopes, subroutines, and so on.

The sequence is not always completely linear; `insn`s are sometimes collected into pairs or triples that correspond to target machine instructions with delay slots. Over a dozen different kinds of (non-*note*) annotations can be attached to an individual `insn`, to identify side effects, specify target machine instructions or registers, keep track of the points at which values are defined and used, automatically increment or decrement registers that are used to iterate over an array, and so on. Insns may also refer to various dynamically allocated structures, including the symbol table.

EXAMPLE 15.20
An RTL insn sequence

A simplified insn sequence for the code of Example C-15.19 appears in Figure C-15.12. The three leading numbers in each insn represent the insn's unique id and those of its predecessor and successor, respectively. The fourth, when present, identifies the insn's basic block. Fields for the various insn annotations are not shown. The :SI *mode specifier* on a memory or register reference indicates access to a single (4-byte) integer; :DI and :QI modes correspond to double (8-byte) and quarter (1-byte) integers.

A full explanation of the RTL notation is beyond what we can cover here. As an example, however, insn 26 loads the memory location found 4 bytes back from the frame pointer (namely, i) into virtual register 64. The following insn, 27, sets the memory location found 8 bytes back from the frame pointer (namely, j) to the result of subtracting register 64 from that same memory location. In parallel (as a side effect), insn 27 also "clobbers" (overwrites) the contents of virtual condition code register 17. ■

In order to generate target code, the back end matches insns against patterns stored in a semiformal description of the target machine. Both this description and the routines that manipulate the machine-dependent parts of an insn are segregated into a relatively small number of separately compiled files. As a result, much of the compiler back end is machine independent, and need not actually be modified when porting to a new machine.

Clang AST format

EXAMPLE 15.21
GCD program as a clang AST

As noted in the main text, the clang front end for LLVM employs a fairly conventional high-level AST format, not unlike gcc's GENERIC. Figure C-15.13 shows a simplified version of the tree for our gcd program. The resemblance to Figure 15.2 is immediately apparent. Unlike the GIMPLE code of Figure reffig-high-gimple, with its explicit gotos, the clang AST nodes encode high-level control structures explicitly. Each CompoundStmt node has one child for every statement in the block; a CallExpr node has one child for the function to be called and one for each argument. Symbol table information is implicit in the declaration nodes of the tree—in this case, FunctionDecl and VarDecl. The cinit annotation on a VarDecl node indicates the presence of a child node to specify the initial value. ■

LLVM IR

LLVM's IR is central to the design of the compiler suite; it has been carefully crafted to represent programs in almost any language, to be easily mapped to almost any modern processor, and to facilitate the full range of modern code improvement techniques. As explained in system documentation (at *llvm.org*), it can be represented, equivalently, as data structures in memory, as compact binary "bitcode," or as human-readable pseudo-assembly notation. Programs are normally passed from one compiler phase to the next as in-memory data structures, but they can also be exported to—or imported from—bitcode files. The pseudo-assembly notation is mainly intended for compiler debugging and development.

```
(insn 5 2 6 2 (set (reg:QI 0 ax) (const_int 0)))
(call_insn 6 5 7 2 (set (reg:SI 0 ax) (call (mem:QI (symbol_ref:DI ("getint"))) (const_int 0))))
(insn 7 6 8 2 (set (reg:SI 60) (reg:SI 0 ax)))
(insn 8 7 9 2 (set (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4))) (reg:SI 60)))
(insn 9 8 10 2 (set (reg:QI 0 ax) (const_int 0)))
(call_insn 10 9 11 2 (set (reg:SI 0 ax) (call (mem:QI (symbol_ref:DI ("getint"))) (const_int 0))))
(insn 11 10 12 2 (set (reg:SI 61) (reg:SI 0 ax)))
(insn 12 11 13 2 (set (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -8))) (reg:SI 61)))
(jump_insn 13 12 14 2 (set (pc) (label_ref 28)))
(barrier 14 13 30)
(code_label 30 14 15 4 4 "")
(insn 16 15 17 4 (set (reg:SI 62) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4)))))
(insn 17 16 18 4 (set (reg:CCGC 17)
                 (compare:CCGC (reg:SI 62) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -8))))))
(jump_insn 18 17 19 4 (set (pc) (if_then_else (le (reg:CCGC 17) (const_int 0)) (label_ref 24) (pc))))
(insn 20 19 21 5 (set (reg:SI 63) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -8)))))
(insn 21 20 22 5 (parallel [
    (set (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4)))
         (minus:SI (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4))) (reg:SI 63)))
    (clobber (reg:CC 17))
]))
(jump_insn 22 21 23 5 (set (pc) (label_ref 28)))
(barrier 23 22 24)
(code_label 24 23 25 6 3 "")
(insn 26 25 27 6 (set (reg:SI 64) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4)))))
(insn 27 26 28 6 (parallel [
    (set (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -8)))
         (minus:SI (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -8))) (reg:SI 64)))
    (clobber (reg:CC 17))
]))
(code_label 28 27 29 7 2 "")
(insn 31 29 32 7 (set (reg:SI 65) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4)))))
(insn 32 31 33 7 (set (reg:CCZ 17)
                      (compare:CCZ (reg:SI 65) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -8))))))
(jump_insn 33 32 34 7 (set (pc) (if_then_else (ne (reg:CCZ 17) (const_int 0)) (label_ref 30) (pc))))
(insn 35 34 36 8 (set (reg:SI 66) (mem/c:SI (plus:DI (reg/f:DI 54) (const_int -4)))))
(insn 36 35 37 8 (set (reg:SI 5 di) (reg:SI 66)))
(call_insn 37 36 40 8 (call (mem:QI (symbol_ref:DI ("putint"))) (const_int 0)))
(insn 40 37 41 8 (clobber (reg/i:SI 0 ax)))
(insn 41 40 39 8 (clobber (reg:SI 59 [ <retval> ])))
(insn 39 41 42 8 (set (reg/i:SI 0 ax) (reg:SI 59 [ <retval> ])))
(insn 42 39 0 8 (use (reg/i:SI 0 ax)))
```

Figure 15.12 Simplified textual RTL for the `gcd` program. Most annotations (more than half the original length) have been elided here. Register 54 is the frame pointer. Local variable `i` is at offset −4. Local variable `j` is at offset −8.

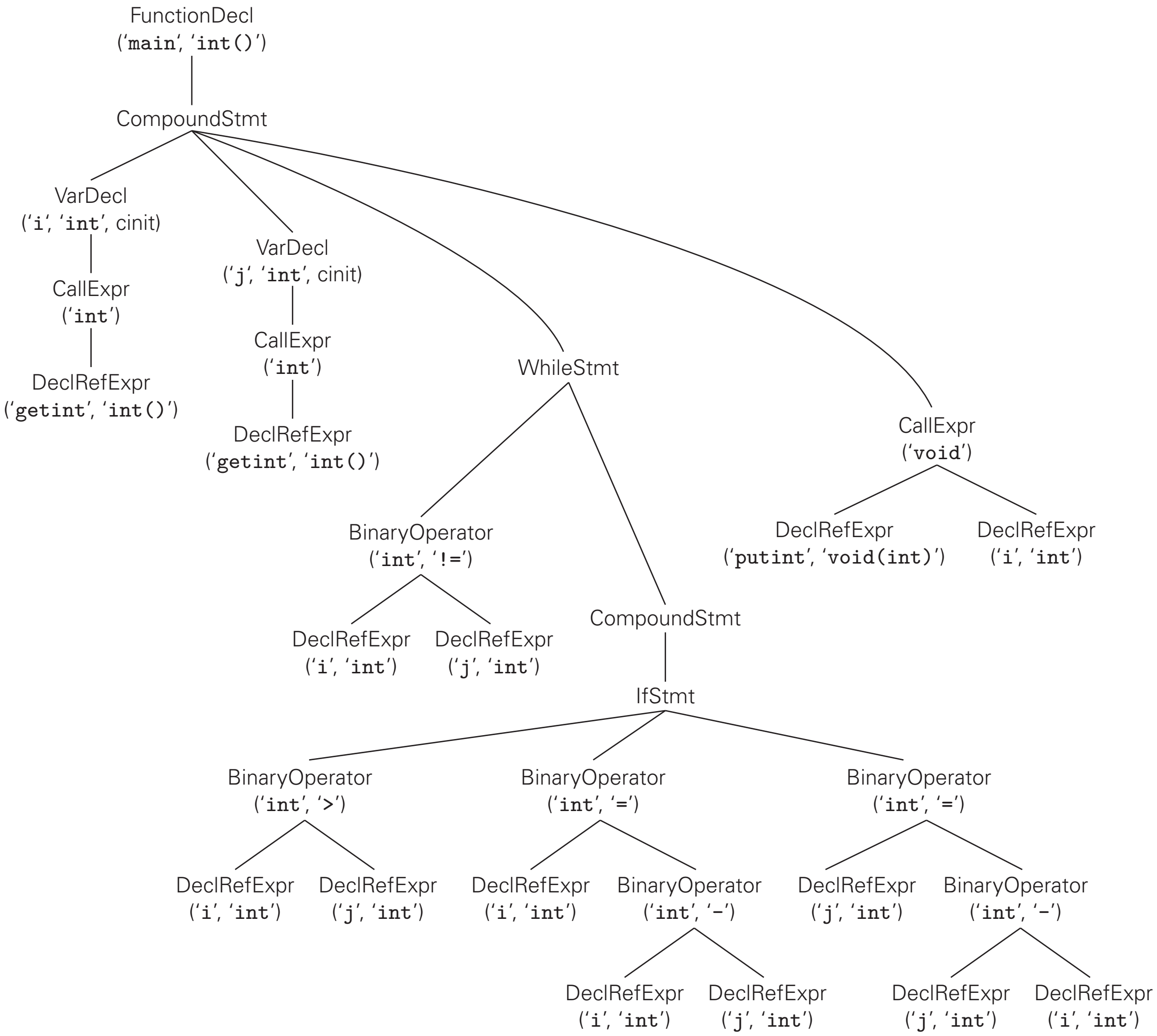


Figure 15.13 Simplified `clang` AST for the `gcd` program. Nodes that implicitly dereference variables (lvalues) to obtain their contents (rvalues) have been elided.

EXAMPLE 15.22
LLVM IR for the GCD program.

LLVM IR for our `gcd` program appears in Figure C-15.14. Unlike the RTL of Figure C-15.12, which was generated at the low, default level of optimization, the code here was generated with a `-O3` command line switch. As a result, variables `i` and `j` are kept in (virtual) registers throughout the computation, rather than being repeatedly read from and written to memory. Because they remain in memory, SSA form requires that we assign the proper incoming values into new virtual registers whenever control paths merge. Specifically, at the top of the loop (label `4`), a `phi` instruction assigns virtual register `%5` (the current location of `j`) the value from

```
define i32 @main() local_unnamed_addr #0 {
  %1 = tail call i32 (...) @getint() #2
  %2 = tail call i32 (...) @getint() #2
  %3 = icmp eq i32 %1, %2
  br i1 %3, label %13, label %4
4:                          ; preds = %0, %4
  %5 = phi i32 [ %11, %4 ], [ %2, %0 ]
  %6 = phi i32 [ %9, %4 ], [ %1, %0 ]
  %7 = icmp slt i32 %5, %6
  %8 = select i1 %7, i32 %5, i32 0
  %9 = sub nsw i32 %6, %8
  %10 = select i1 %7, i32 0, i32 %6
  %11 = sub nsw i32 %5, %10
  %12 = icmp eq i32 %9, %11
  br i1 %12, label %13, label %4, !llvm.loop !6
13:                         ; preds = %4, %0
  %14 = phi i32 [ %1, %0 ], [ %9, %4 ]
  tail call void @putint(i32 noundef %14) #2
  ret i32 0
}
```

Figure 15.14 **Textual LLVM IR for function `main` in the `gcd` program,** generated at optimization level `-O3`. Comments begin with a semicolon; metadata begins with an exclamation point. (Function calls are labeled `tail` not because they are actually tail recursive, but because they do not violate any of the rules that would prevent reuse of the stack frame if they *were* tail recursive.)

register `%2` if control has entered from the header block, and from register `%11` if control has come around from the bottom of the loop. A second `phi` instruction makes a similar choice for `i` at the top of the loop, and a third for `i` at the beginning of the footer. Among other things, the fact that every virtual register has only one assignment in the code means that instructions can safely be reordered so long as operands are always computed before they are used. We never have to worry about "overwriting" a virtual register before its final use, or about writing values to it out of order.

In addition to promoting `i` and `j` to registers, `-O3` optimization causes the compiler to use *predication* (Section C-5.3.2), rather than branching, for the `if` … `then` … `else` in the loop. Specifically, the `icmp` instruction at the top of the loop assigns the outcome of the `if` comparison into virtual register `%7`. The `select` instructions then use this register to chose whether to place a zero or one of `j` and `i`, respectively, into registers `%8` and `%10`. Finally, the `sub` instructions subtract these values from `i` and `j`, effectively updating one and leaving the other unchanged.

CHECK YOUR UNDERSTANDING

24. Characterize GIMPLE, RTL, `clang` AST format, LLVM IR, Java bytecode, and Common Intermediate Language as high-, medium-, or low-level intermediate forms.

25. Name three languages (other than C) for which there exist `gcc` front ends.

26. Name three languages (other than C) for which there exist LLVM front ends but not `gcc` front ends.

27. What is the internal IF of `gcc`'s front ends?

28. Give brief descriptions of GIMPLE and RTL. How do they differ? Why was GIMPLE introduced?

29. Compare RTL and LLVM IR. In what ways does the latter more closely resemble typical assembly language?

15 Building a Runnable Program

15.7 Dynamic Linking

To be amenable to dynamic linking, a library must either (1) be located at the same address in every program that uses it, or (2) have no relocatable words in its code segment, so that the content of the segment does not depend on its address. The first approach is straightforward but restrictive: it generally requires that we assign a unique address to every sharable library; otherwise we run the risk that some newly created program will want to use two libraries that have been given overlapping address ranges. In Unix System V R3, which took the unique-address approach, shared libraries could only be installed by the system administrator. This requirement tended to limit the use of dynamic linking to a relatively small number of popular libraries. The second approach, in which a shared library can be linked at any address, requires the generation of *position-independent code*. It allows users to employ dynamic linking whenever they want, without administrator intervention.

The cost of user-managed dynamic linking is that executable programs are no longer self-contained. They depend for correct execution on the availability of appropriate dynamic libraries at execution time. If different programs are built with different expectations of (which versions of) which libraries will be available, conflicts can arise. On Microsoft platforms, where dynamic libraries have names ending in `.dll`, compatibility problems are sometimes referred to as "DLL hell." The frequency and severity of the problem can be minimized with good software engineering practice. In particular, a *package management system* may maintain a database of dependences between programs and libraries, and among the libraries themselves. If installer programs use the database correctly, problems will be detected at install time, when they can reasonably be addressed, rather than at the arbitrarily delayed point at which a program first attempts to use an incompatible or missing library.

15.7.1 Position-Independent Code

A code segment that contains no relocatable words is said to constitute *position-independent code* (PIC). To generate PIC, the compiler must observe the following rules:

1. Use PC-relative addressing, rather than jumps to absolute addresses, for all internal branches.
2. Similarly, avoid absolute references to statically allocated data, by using displacement addressing with respect to some standard base register. If the code and data segments are guaranteed to lie at a known offset from one another, then the program counter can be used for this purpose. Otherwise, the caller must initialize some other base register as part of the entry point's calling sequence.
3. Use an extra level of indirection for every control transfer out of the PIC segment, and for every load or store of static memory outside the corresponding data segment. The indirection allows the (non-PIC) target address to be kept in the data segment, which is private to each program instance.

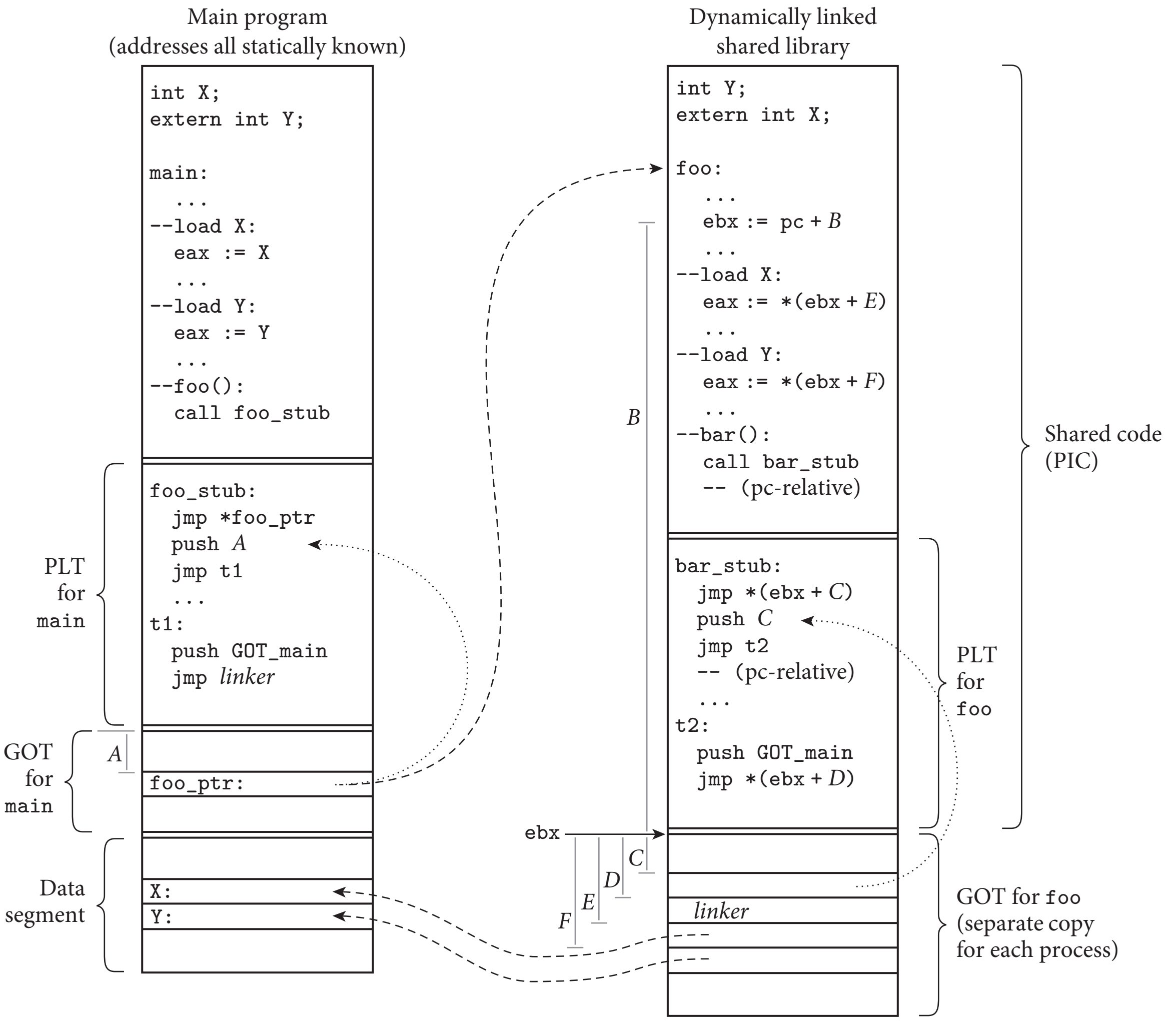
EXAMPLE 15.23
PIC under x86/Linux

Exact details vary among processors, vendors, and operating systems. Conventions for `gcc` on recent versions of x86 Linux are illustrated in Figure C-15.15. Each code segment is accompanied by a *linkage table*—known in Linux as the segment's *global offset table* (GOT). This table lists the locations of all code and data whose addresses were not statically determined. All processes that use the same library share a single copy of the library's code segment, but each process has its own copy of the library's GOT. Both the code segment and the GOT can lie at different locations in the address spaces of different processes, but the *offset* between the two must always be the same.

Like the main program, each shared library is typically composed of multiple compilation units, joined together by a *static linker*, which resolves internal references. Resolution of references from the main program into shared libraries—or among the libraries themselves—is delayed until load time or run time, and is the job of the *dynamic linker*. By construction, shared libraries never make references back into the main program.

Libraries are permitted to have (process-private) data as well as code, but the total amount of such data is assumed (in Linux, at least) to be small enough that the data can be statically linked without wasting significant space. Each process therefore has a single data segment (shown in the figure at the lower left), containing the data of the main program and of all the libraries it may call, directly or indirectly. (Extensions to delay the linking of library data are considered in Exercise C-15.14.)

Focusing for the moment on the dashed arrows of the figure (and ignoring the dotted arrows), a read of `X` or `Y` in `main` can use a statically resolved address. A read of `X` or `Y` in `foo` uses PC-relative addressing to find the appropriate slot in `foo`'s GOT, and then loads `X` or `Y` indirectly. Similar indirection is required for subroutine calls into dynamically linked libraries. To avoid duplication of the indirection code, the compiler incorporates a (shared, read-only) `procedure`



```
int X;
extern int Y;

main:
  ...
--load X:
  eax := X
  ...
--load Y:
  eax := Y
  ...
--foo():
  call foo_stub
```

```
foo_stub:
  jmp *foo_ptr
  push A
  jmp t1
  ...
t1:
  push GOT_main
  jmp linker
```

```
foo_ptr:
```

```
X:
Y:
```

```
int Y;
extern int X;

foo:
  ...
  ebx := pc + B
  ...
--load X:
  eax := *(ebx + E)
  ...
--load Y:
  eax := *(ebx + F)
  ...
--bar():
  call bar_stub
  -- (pc-relative)
```

```
bar_stub:
  jmp *(ebx + C)
  push C
  jmp t2
  -- (pc-relative)
  ...
t2:
  push GOT_main
  jmp *(ebx + D)
```

Figure 15.15 **A dynamically linked shared library.** Calls to foo and bar are made indirectly, using an address stored in the global offset tables (GOTs) of main and foo, respectively. Similarly, references to global variables X and Y, when made from foo, must employ a level of indirection. Resolved values are shown with dashed lines; initial values to support lazy linking (Section C-15.7.2) are shown with dotted lines. In the prologue of foo, register ebx is set to point to foo's GOT, using pc-relative arithmetic.

linkage table (PLT) in each code segment. To effect a call to foo, main calls a *stub* routine, here named foo_stub. This, in turn, performs an indirect jump to the address of foo found in main's GOT. Inside foo, the call to bar is only slightly more complicated: the compiler must use PC-relative addressing to find the appropriate slot in foo's GOT. ■

EXAMPLE 15.24

PC-relative addressing on the x86

Most machines—including the x86—can perform branches and calls using PC-relative addressing. In our Linux example (Figure C-15.15), the machine-language encoding of `call bar_stub` in library `foo` will specify the offset between the call instruction and the `bar_stub` location in `foo`'s PLT.

Many machines can also use PC-relative addressing in load and store instructions. On the x86-64, for example, the load of `X` in `foo` could say `rax := *(rip + G)`, where *G* is the offset from the load instruction to `X`'s entry in `foo`'s GOT (on the x86-64, `rip` [instruction pointer register] is the name of the program counter). Unfortunately, the x86-32 does not support PC-relative addressing for loads and stores. To compensate, each PIC code segment on x86-32 Linux defines the following tiny subroutine:

```
get_pc:
    ebx := *esp     -- load location referred to by esp
    ret                 -- i.e., the return address -- into ebx
```

Given this definition, the pseudo-instruction `ebx := pc + B` in Figure C-15.15 can be implemented as

```
    call get_pc
    ebx += B
```

after which `ebx` can be used as the base for displacement addressing within `foo`'s GOT.

15.7.2 Fully Dynamic (Lazy) Linking

If all or most of the symbols exported by a shared library were always referenced by the parent program, it might make sense to link the library in its entirety at load time. When the program began running, its GOTs would then appear as suggested by the dashed arrows in Figure C-15.15. Certain systems indeed work in this fashion. In any given execution of a program, however, there may be references to libraries that are not actually used, because the input data never cause execution to follow the code path(s) on which the references appear. If these "potentially unnecessary" references are numerous, we may avoid a significant amount of work by linking the library *lazily* on demand. Moreover even in a program that uses all its symbols, incremental lazy linking may improve the system's interactive responsiveness by allowing programs to begin execution faster. A language system that allows the dynamic creation or discovery of program components (e.g., as in Common Lisp or Java) must also use lazy linking to delay the resolution of external references in dynamically compiled components.

EXAMPLE 15.25

Dynamic linking in Linux on the x86

When a Linux program first starts running, the data entries in its GOTs are indeed initialized as previously discussed; all addresses are known, because data locations are statically assigned. Code entries in the GOTs, however, point back into the corresponding PLTs, as suggested by the dotted arrows in Figure C-15.15.

Now consider what happens when `main` calls `foo_stub`. The `foo_ptr` entry in `main`'s GOT points to the second instruction of `foo_stub`—immediately after the indirect jump. That jump, in other words, ends up targeting the very next instruction, as if it had not happened at all. The next instruction, for its part, pushes onto the stack the offset of `foo`'s entry in `main`'s GOT. It then jumps (using PC-relative addressing) to a special entry in the PLT. This entry in turn pushes the address of `main`'s GOT and jumps to the dynamic linker, whose address is statically known. The dynamic linker consults symbol table information found in `main`'s executable file. Specifically, it looks up the GOT address and offset that were passed to it on the stack and discovers that they correspond to `foo`. It chooses a place for `foo` in the process's address space, creates a (process-specific) `foo` GOT at the appropriate offset (using symbol table information from `foo`'s own object file), initializes any data locations in that GOT to point to appropriate locations in the process's data segment, and initializes code locations in the GOT to point to the second instructions of the corresponding entries in `foo`'s PLT.

Now that `foo` has been given a location in the process's address space, the dynamic linker can modify `foo`'s entry in `main`'s PLT so that subsequent calls from `main` to `foo` will skip the linking step, and instead follow the single indirection suggested by the dashed arrow in Figure C-15.15. Last of all, the linker pops its arguments from the stack (leaving the return address pushed by `main` in its original call to `foo_stub`) and branches directly to `foo`. When `foo` completes, it will return to the correct address in `main`.

If and when `foo` calls `bar`, a similar series of events will take place. The principal difference is that both the body of `foo` and the stubs in its PLT must use PC-relative addressing to access entries in `foo`'s GOT. ■

CHECK YOUR UNDERSTANDING

30. Explain the addressing challenge faced by dynamic linking systems.

31. What is *position-independent code*? What is it good for? What special precautions must a compiler follow in order to produce it?

32. Explain the need for PC-relative addressing in position-independent code. How is it accomplished on the x86-32?

33. What is the purpose of a *linkage table*?

34. What is *lazy* dynamic linking? What is its purpose? How does it work?

15 Building a Runnable Program

15.9 Exercises

15.12 Compare and contrast GIMPLE with the notation we have been using for abstract grammars (Section 4.1).

15.13 PC-relative branches on many processors are limited in range—they can only target locations within 2^k bytes of the current PC, for some k less than the wordsize of the machine. Explain how to generate position-independent code that needs to branch farther than this.

15.14 We have noted that Linux creates a single data segment containing all the static data of libraries that might be called (directly or indirectly) by a given program. The space required for this segment is usually not a problem: most libraries have little static data—often none at all. Suppose this were not the case. If we wanted to perform dynamic linking for modules with large amounts of per-module static data, how could we extend Linux's dynamic linking mechanisms to perform fully dynamic (lazy) linking not only of code, but also of data?

15.15 In Example C-9.72 we described how the GNU Ada Translator (`gnat`) for the x86 uses dynamically generated code to represent a subroutine closure. Explain how a similar technique could be used to simplify the mechanism of Figure C-15.15, if we were willing to modify code segments at run time.

15 Building a Runnable Program

15.10 Explorations

15.21 Find the on-line documentation for gcc, which explains both GIMPLE and RTL, and enumerates command-line flags that will cause the compiler to dump its intermediate forms to standard output. (Version 4.8.4 of the compiler supports 26 such flags for GIMPLE and 67 for RTL.) Using appropriate flags and a small but nontrivial input program, arrange for the compiler to dump several versions of both GIMPLE and RTL. Study the output and describe how it has been changed by the intervening code improvement phases.

15.22 Find out how linking works under your favorite non-Linux system. Can code be dynamically linked? Can (nonprivileged) users create shared libraries? How does the loader or dynamic linker determine which libraries a program will need? How does it locate their object code? If your compiler can generate both position-independent and non-position-independent code, how do the two compare in size and run-time efficiency?

15.23 Learn about *pointer swizzling* [Wil92], originally developed to run programs on machines with insufficient virtual address space. Explain its connection to dynamic linking.

15.24 Learn about ASIS, the Ada Semantic Interface Specification. How does it improve on tools based on the earlier Diana notation? How does it work in gnat?

15.25 Learn about MLIR, the Multi-Level Intermediate Representation being developed as an extension to the LLVM framework. In contrast to LLVM IR, MLIR aims to represent programs at multiple levels of abstraction, and in particular to facilitate high-level optimization for GPUs, tensor units, and other specialized accelerators. Explore the origins of MLIR in Google's TPU project. What exactly does MLIR enable that traditional medium- and low-level IFs do not?

16 Run-time Program Management

16.1.2 The Common Language Infrastructure

Work on the system that became the Common Language Infrastructure (CLI) began at Microsoft Corporation in the late 1990s, and was able to benefit from experience with Java and the JVM, which were already well established. The most significant differences between the virtual machines, however, stem from Microsoft's emphasis on cross-language interoperability—an emphasis that predates the JVM by many years.

Growing out of earlier work on the DDE, OLE, COM, ActiveX, and DCOM projects, the beta version of .NET was released in 2000. In addition to a virtual machine, it includes libraries, servers, and tools for a wide variety of local and distributed services, including user interface management, database access, networking, and security. A specification for the virtual machine—the CLI—was standardized by ECMA in 2001 and by the ISO in 2003. The standard has been updated several times over the years; version 6 was released in June 2012 [Int12a].

Perhaps the most significant contribution of the CLI is the definition of a Common Type System (CTS) for all supported languages. Encompassing nearly everything described in Chapters 8 and 10 of this book, the CTS provides a superset of what any particular language needs, while requiring common semantics and implementation wherever the type systems of more than one language intersect. In addition to the CTS, the CLI defines a virtual machine architecture, the VES (Virtual Execution System); an instruction set for that machine, the CIL (Common Intermediate Language); and a portable file format for code and metadata, PE (Portable Executable) assemblies.

C# is in some sense the premier language for .NET, and was developed concurrently with it. Several dozen languages have been ported to the CLI, however, and several of these, including Visual Basic, C++/CLI (formerly Managed C++), and F# (a descendant of OCaml) are now in widespread use.

Thanks to the ECMA/ISO standard, it is possible for organizations other than Microsoft to build implementations of the CLI. The leading such implementation is the open-source Mono project, led by Xamarin, Inc. (a Microsoft subsidiary). Mono runs on a wide variety of platforms, but tends to lag slightly behind .NET in the addition of new features. Outside Microsoft, Java and the JVM still dominate. Within Microsoft, most new development today employs C#. Microsoft calls its CLI implementation the Common Language Runtime (CLR).

Architecture and Comparison to the JVM

In many ways, the CLI resembles the JVM. Both systems define a multithreaded, stack-based virtual machine, with built-in support for garbage collection, exceptions, virtual method dispatch, and interface inheritance. Both represent programs using a platform-independent, self-descriptive, bytecode notation. For languages like C#, the CLI provides all the safety of the JVM, including definite assignment, strong typing, and protection against overflow or underflow of the operand stack.

The biggest contrasts between the JVM and CLI stem from the latter's support for multiple programming languages (the following is not a comprehensive list).

Richer Type System The Common Type System (discussed below) supports both value and reference variables of structured types (the JVM is limited to references). The CTS also has true multidimensional arrays (allocated, contiguously, as a single operation); function pointers; explicit support for generics; and the ability to enforce structural type equivalence.

Richer Calling Mechanisms To facilitate the implementation of functional languages, the CLI provides explicit tail-recursive function calls (Section 6.6.1);

DESIGN & IMPLEMENTATION

16.7 Assuming a just-in-time compiler

Like the JVM, the CLI has behavior defined in terms of an abstract virtual machine. Where Java's virtual machine may in practice be either interpreted or just-in-time compiled, however, the CLI was designed from the outset for just-in-time compilation. Several minor differences between the virtual machines reflect this difference in expected implementations. Arithmetic instructions in Java bytecode generally include an explicit indication of operand type: there are, for example, four separate opcodes for 32- and 64-bit integer and floating-point addition. In the CLI's Common Intermediate Language (CIL), there is only one `add` instruction: it figures out what to do based on the types of its operands. In type-safe code, of course, the type of every operand is statically known, and either a compiler or an interpreter can inspect the types of arguments and figure out what to do. The compiler, however, only has to do this once, at compile time; the interpreter has to do it every time it encounters the instruction. In a similar vein, slots in the local variable array of the CLI VES can be of arbitrary size, and are required to hold a value of a single, statically known type throughout the execution of the method. For the sake of space efficiency and rapid indexing, the JVM reserves exactly 32 bits for every slot (`long`s and `double`s take two consecutive slots), and a given slot can be used for values of different types at different points in time.

these discard the caller's frame while retaining the dynamic link. The CLI also supports both value and reference parameters, variable numbers of parameters (in the fully general sense of C), multiple return values, and nonvirtual methods, all of which the JVM lacks.

Unsafe Code For the benefit of C, C++, and other non-type-safe languages, the CLI supports explicitly unsafe operations: nonconverting type casts, dynamic allocation of non-garbage-collected memory, pointers to non-heap data, and pointer arithmetic. The CLI distinguishes explicitly between *verifiable* code, which cannot use these features, and *unverifiable* code, which can. (Verifiable code must also follow a host of other rules.)

Miscellaneous Again for the sake of multiple languages, the CLI supports global data and functions, local variables whose shapes and sizes are not statically known, optional detection of arithmetic overflow, and rich facilities for "scoped" security and access control.

As in the JVM, every CLI thread has a small set of base registers and a stack of method call frames, each of which contains an array of local variables and an operand stack for expression evaluation. Each frame also contains a local memory pool for variables of dynamic and elaboration-time shape. Incoming parameters have their own separate space in the CLI; in the JVM they occupy the first few slots of the local variable array.

The Common Type System

The VES and CIL provide instructions to manipulate data of certain built-in types. A few additional types are predefined, and have built-in names in CLI metadata. To these, the CTS adds a wide variety of type constructors. For each, it defines both behavior *and* representation. No single language provides all the types of the CTS, but (with occasional compromises) each provides a subset.

The Common Language Specification (CLS) defines a subset of the CTS intended for cross-language interaction. It omits several type constructors provided by the CTS, and places restrictions on others. Standard libraries (collection classes, XML, network support, reflection, extended numerics) restrict themselves (with occasional exceptions) to types in the CLS. Not all languages support the full CLS; code written in those languages cannot make use of library facilities that require unsupported types.

Built-in Types The VES and CIL provide instructions to manipulate the following types:

- Integers in 8-, 16-, 32-, and 64-bit lengths, both signed and unsigned
- "Native" integers, of the length supported by the underlying hardware, again both signed and unsigned
- IEEE floating point, both single and double precision
- Object references and "managed" pointers

Managed pointers are different from references: while typed, they don't necessarily point to the beginning of a dynamically created object. Specifically, they can refer to fields within an object or to data outside the heap. The CIL makes sure these pointers are known to the garbage collector, which must avoid reclaiming any object *O* when a managed pointer refers to a field inside *O*. More details on pointers and references can be found in Sidebar C-16.8.

Beyond the basic hardware-level types, CLI metadata treats Booleans, characters, and strings as built-ins. Booleans and characters are manipulated in the VES using instructions intended for short integers; strings are manipulated by accessing their internal structure.

Constructed Types To the built-in types, the CTS adds the following:

Dynamically allocated instances of class, interface, array, and delegate types. These are the things to which references (the built-in type) can refer. Arrays can be multidimensional, and are stored in row-major order. Delegates are closures (subroutine references paired with referencing environments).

Methods — function types.

Properties — getters and setters for objects.

Events — lists of delegates, associated with an object, that should be called in response to changes to the object.

Value types — records (structures), unions, and enumerations.

Boxed value types — values embedded in a dynamically allocated object so that one can create references to them.

Function pointers — references to static functions: type-safe, but without a referencing environment.

Typed references — pointers bundled together with a type descriptor, used for C-style variable argument lists.

Unmanaged pointers — as in C, these can point to just about anything, and support pointer arithmetic. They *cannot* point to garbage-collectible objects (or parts of objects) in the heap.

With these type constructors come extensive semantic rules, covering such topics as identity and equality,[1] casting and coercion, scoping and visibility, interface inheritance, hiding and overriding of members, memory layout, initialization, type safety, and verification. The details occupy hundreds of pages in the CLI documentation.

The Common Language Specification Because no single language implements the entire CTS, one cannot use arbitrary CTS types in a general-purpose interface intended for use from many different languages. The Common Language Specification (CLS) defines a subset of the CTS that most (though not all) languages

[1] These are reminiscent of the relationships discussed in Sections 7.5 and 11.3.3.

can accommodate. Among other things, it omits several of the types provided by the CTS, including signed 8-bit integers; unsigned native, 16-, 32-, and 64-bit integers; boxed value types; global static fields and methods; unmanaged pointers; typed references; and methods with variable numbers and types of arguments. The CLS also imposes a variety of restrictions on the use of other types. It establishes naming conventions, limits the use of overloading, and defines the operators and conversions that programs can assume are supported on built-in types. It requires a lower bound of zero on each dimension of array indexing. It prohibits fields and static methods in interfaces. It insists that a constructor be called exactly once for each created object, and that each constructor begin with a call to a constructor of its base class. None of these restrictions applies to program components that operate only within a given language.

Generics As described in Section C-7.3.5, generics were added to Java and C# in very different ways. Partly to avoid the need to modify the JVM, Java generics were defined in terms of *type erasure*, which effectively converts all generic types to `Object` before generating bytecode. C# generics were defined in terms of *reification*, which creates a new concrete type every time a generic is instantiated with different arguments. Reified generics have been supported directly by the CLI since .NET

DESIGN & IMPLEMENTATION

16.8 References and pointers

The reference and pointer types of the CTS are a source of potential confusion. In a language like Java, reference types provide the only means of indirection. They refer to dynamically allocated instances of class, interface, and array types. Managed pointers provide additional functionality for languages like C# and Microsoft's C++/CLI, which permit references to the insides of objects and to values outside the CLI heap. Managed pointers are understood by the garbage collector, and can be used in type-safe code: If a managed pointer p refers to a field of object O, then the collector will know that O is live. It will also update p automatically whenever it moves O.

Unmanaged pointers exist for the sake of languages like C. They are incompatible with garbage collection, and cannot point to objects in the heap. They are also incompatible with type safety, and cannot be used in verifiable code.

Typed references (`typedrefs`) in the CLI include the information needed to correctly manipulate references to values (e.g., in variable argument lists) whose type cannot be statically determined.

Version 2.0 of the CLI introduced *controlled-mutability* managed pointers (also known, somewhat inaccurately, as *read-only* pointers). Operations on these pointers are constrained to prevent modification of the referenced object. Read-only pointers are used in boxing and array contexts where generics require the ability to generate a pointer to data of a value type, but modification of that data might not be safe.

version 2.0, introduced by Microsoft in 2005 and codified by ECMA and ISO in 2006.

EXAMPLE 16.39

Generics in the CLI and JVM

Reified generic types are fully described in CLI metadata, allowing full type checking and reflection. Consider the following code in C#:

```
class Node<T> {
    public T val;
    public Node<T> next;
}
...
Node<int> n = new Node<int>();
Console.WriteLine(n.GetType().ToString());
```

If Node is an outermost class, the final line will print Node`1[System.Int32]. The equivalent code in Java (running on the JVM) will simply print class Node. To support generics, CLI version 2 extended the rules for type compatibility and verification, and introduced new versions of several CIL instructions.

Metadata and Assemblies

Portable Executable (PE) *assemblies* are the rough equivalent of Java .jar files: they contain the code for a collection of CLI classes. PE is based on the Common Object File Format (COFF), originally developed for AT&T's System V Unix. It is the native object file format for Windows systems, extended to accommodate CIL as an optional instruction set. Given the requirements of native-code executable files (e.g., relocation—see Section 15.4), PE is quite a bit more complicated than Java .class and .jar format. A PE assembly contains a general-purpose PE header, a special CLI header, metadata describing the assembly's types and methods, and CIL code for the methods.

The metadata of an assembly has a complex internal structure. (A diagram of the interconnections among some two dozen different kinds of tables fills two pages of the annotated CLI standard [MR04, pp. 322–323].) The metadata begins with a *manifest* that specifies the files included and directly referenced, the types exported and imported, versioning information, and security permissions. This is followed by descriptions of all the types, and signatures for all the methods. Unlike the Java constant pool, the metadata of an assembly is not directly visible to the assembly's code; it may be rearranged by the JIT compiler in implementation-dependent ways, so long as it remains available to reflection routines at run time (obviously, those routines are also implementation dependent).

The Common Intermediate Language

Just as the CLI VES bears a strong resemblance to the JVM, CIL bears a strong resemblance to Java bytecode. Version 6 of the ECMA standard defines some 219 instructions, most with single-byte opcodes. Most instructions take their arguments from, and return results to, the operand stack of the current method frame. Others take explicit arguments representing variables, types, or methods.

Java bytecode and CIL are similarly dense—they require roughly the same number of bytes per instruction on average.

Many of the differences between the two intermediate languages are essentially trivial. Java bytecode is big-endian; CIL is little-endian. Java bytecode has explicit instructions for monitor entry and exit; these are method calls in the CLI. CIL allows arbitrary offsets for branches; Java bytecode limits them to 64K bytes.

A few more significant differences stem from the assumption that CIL will always be JIT-compiled, as described in Sidebar C-16.7. The most obvious difference here is that Java bytecode encodes type information explicitly in opcodes, while CIL requires it to be inferred from arguments. CIL also includes an explicit instruction (`ldtoken`) that will push a "run-time handle" for a method, type, or field. While the metadata of a CIL assembly must all be available at run time, its format may be implementation dependent; the JIT compiler translates `ldtoken` into machine code consistent with that format. In the JVM, the class file constant pool is assumed to be available at run time, in its standard format; an ordinary "load constant" instruction suffices to push the desired reference.

A more subtle difference is the separation of arguments from local variables in the CLI (they share one array in the JVM). Separate arrays admit special one-byte load instructions for both the first few arguments and the first local variables, without requiring that they have interleaved slots; this in turn may make it easier to generate object code in which arguments occupy contiguous locations in memory (as, for example, in the argument build area of the stack described in Section C-9.2.2).

Finally, as already suggested, several features of CIL, not found in Java bytecode, stem from the need to support multiple source languages. We have noted that the CLI provides value types, reference parameters, and optional overflow checking on arithmetic; all of these are reflected in the CIL instruction set. There are also several extra ways to make subroutine calls. Where Java bytecode supports only static, virtual, and dynamic method invocations, CIL has (1) nonvirtual method calls, as in C++ (these implicitly pass `this`, as virtual calls do); (2) indirect calls (i.e., calls through function pointers); (3) tail calls, which discard the caller's frame; and (4) *jumps*, which redirect control to a method after executing some optional prologue (e.g., for `this` pointer adjustment in languages with multiple inheritance; see Section C-10.6).

EXAMPLE 16.40

CIL for a list insert operation

To illustrate CIL, let us return to the linked-list set of Example **??**. The declarations given there are valid in both Java and C#. The `insert` method for this class appears in Figure C-16.7. C# source (which is again identical to the Java version) is on the left; a symbolic representation of the corresponding CIL is on the right. As in Example **??**, there are many examples of special one-byte load and store instructions (here specified with a *.index* suffix on the opcode), and of instructions that operate implicitly on the operand stack. ■

Verification As we have noted, the CLI distinguishes between *verifiable* and *unverifiable* code. Verifiable code must satisfy a large variety of constraints that guarantee type safety and catch many common programming errors. In particular,

```
                                 | .method private hidebysig
public void insert(int v) {      |        instance default void insert (int32 v)  cil managed
                                 | {
                                 |     // Method begins at RVA 0x2070      // RVA == relative
                                 |     // Code size 108 (0x6c)             //     virtual address
                                 |     .maxstack 3
                                 |     .locals init (
                                 |         class LLset/node    V_0,        // n
                                 |         class LLset/node    V_1)        // t
    node n = head;               |     IL_0000:  ldarg.0
                                 |     IL_0001:  ldfld class LLset/node LLset::head
                                 |     IL_0006:  stloc.0
                                 |     IL_0007:  br IL_0013        // jump to header of rotated loop
                                 |     IL_000c:  ldloc.0           // n -- beginning of loop body
                                 |     IL_000d:  ldfld class LLset/node LLset/node::next
                                 |     IL_0012:  stloc.0           // n = n.next
    while (n.next != null        |     IL_0013:  ldloc.0           // n -- beginning of loop test
           && n.next.val < v) {  |     IL_0014:  ldfld class LLset/node LLset/node::next
                                 |     IL_0019:  brfalse IL_002f   // exit loop if n null
                                 |     IL_001e:  ldloc.0           // n
                                 |     IL_001f:  ldfld class LLset/node LLset/node::next
                                 |     IL_0024:  ldfld int32 LLset/node::val
        n = n.next;              |     IL_0029:  ldarg.1           // v
    }                            |     IL_002a:  blt IL_000c       // continue loop
    if (n.next == null           |     IL_002f:  ldloc.0           // n
        || n.next.val > v) {     |     IL_0030:  ldfld class LLset/node LLset/node::next
                                 |     IL_0035:  brfalse IL_004b
                                 |     IL_003a:  ldloc.0           // n
                                 |     IL_003b:  ldfld class LLset/node LLset/node::next
                                 |     IL_0040:  ldfld int32 LLset/node::val
                                 |     IL_0045:  ldarg.1           // v
                                 |     IL_0046:  ble IL_006b
        node t = new node();     |     IL_004b:  newobj instance void class LLset/node::'.ctor'()
                                 |     IL_0050:  stloc.1           // t
        t.val = v;               |     IL_0051:  ldloc.1           // t
                                 |     IL_0052:  ldarg.1           // v
                                 |     IL_0053:  stfld int32 LLset/node::val
        t.next = n.next;         |     IL_0058:  ldloc.1           // t
                                 |     IL_0059:  ldloc.0           // n
                                 |     IL_005a:  ldfld class LLset/node LLset/node::next
                                 |     IL_005f:  stfld class LLset/node LLset/node::next
        n.next = t;              |     IL_0064:  ldloc.0           // n
                                 |     IL_0065:  ldloc.1           // t
                                 |     IL_0066:  stfld class LLset/node LLset/node::next
    } // else v already in set   |     IL_006b:  ret
}                                | } // end of method LLset::insert
```

Figure 16.7 **C# source and CIL for a list insertion method.** Output on the right was produced by the Mono project's `mcs` (compiler) and `monodis` (disassembler) tools, with additional comments inserted by hand. Note that the compiler has rotated the test to the bottom of the `while` loop, which occupies lines `IL_000c` through `IL_002a` in the output code.

the VES can be sure that a verifiable program will never access data outside its logical address space. Among other things, this guarantee ensures fault containment for verifiable modules that share a single physical address space.

Unverifiable code can make use of unsafe language features (e.g., unions and pointer arithmetic in C), but must still conform to more basic rules for validity (well-formedness) of CIL. Together, the components of the VES (i.e., the JIT compiler, loader, and run-time libraries) *validate* all loaded assemblies, and *verify* those that claim to be verifiable. Any standard-conforming implementation of the CLI must run all verifiable programs. Optionally, it may also run validated but not verifiable programs.

As in the JVM, verification requires data flow analysis to check type consistency and lack of underflow and overflow in the operand stack. The CLI standard requires verifiable routines to specify that all local variables are initialized to zero. CLI implementations typically perform definite assignment data flow analysis anyway, to identify cases in which those initializations can safely be omitted. The standard also requires numerous checks on individual instructions. Many of these are also performed by the JVM. Local variable references, for example, are statically checked to make sure they lie within the declared bounds of the stack frame. Other checks stem from the presence of unsafe features in the CLI. Verifiable code cannot use unmanaged pointers or unions, for example, nor can it perform most indirect method calls.

✓ CHECK YOUR UNDERSTANDING

38. Summarize the architecture of the Common Language Infrastructure. Contrast it with the JVM. Highlight those features intended to facilitate cross-language interoperability.

39. Describe how the choice of just-in-time compilation (and the rejection of interpretation) influenced the structure of the CLI.

40. Describe several different kinds of references supported by the CLI. Why are there so many?

41. What is the purpose of the Common Language Specification? Why is it only a subset of the Common Type System?

42. Describe the CLI's support for *unsafe* code. How can this support be reconciled with the need for safety in embedded settings?

16 Run-time Program Management

16.5 Exercises

16.14 Using Oracle's `jaotc` compiler and `mono --aot`, compile the code of Figures 16.2 and C-16.7 all the way to machine language. Disassemble and compare the results. Can all the differences be attributed to variations in the quality of the compilers, or are any reflective of more fundamental differences between the source languages or virtual machines?

16.15 Rewrite the list insertion method of Example C-16.40 in F# instead of C#. Compile to CIL and compare to the right side of Figure C-16.7. Discuss any differences you find.

16.16 Building on the previous exercise, rewrite your list insertion routine (both C# and F# versions) to be generic in the type of the list elements. Compare the generic and nongeneric versions of the resulting CIL and discuss the differences.

16.17 Extend your F# code from Exercise C-16.16 to include list removal and search routines. After finding and reading appropriate documentation, package these routines in a library that can be called in a natural way not only from F# but also from C#.

16 Run-time Program Management

16.6 Explorations

16.26 Learn the details of the CLI verification algorithm (Partition III, Section 1.8 of the ECMA standard, version 4 [Int12a]). Pay particular attention to the rules for *merging* compatible types at joins in the control flow graph, and for dealing with generics.

16.27 Learn more about the .NET Language-Integrated Query mechanism (LINQ), mentioned in Example 16.29. Discuss its use of attributes. Write a program that uses it to interface to a database through SQL. Write another program that uses it to process the elements of a set from the `System.Collections` library.

16.28 Like most scripting languages, Perl 5 compiles its input to an internal syntax tree format, which it then interprets. Explore this implementation, and characterize the circumstances under which the interpreter may need to call back into the compiler during execution. Also explore the `perlcc` command-line script (itself written in Perl), which translates source code to either bytecode or machine code.

In several cases, the interpreter may need to call back into the compiler during execution. Features that force such dynamic compilation include `eval`, which compiles and then interprets a string; `require`, which loads a library package; and the `ee` version of the substitution command, which performs expression evaluation on the replacement string:

```
$foo = "abc";
$foo =~ s/b/2 + 3/ee;       # replace b with the value of 2 + 3
print "$foo\n";             # prints a5c
```

Perl can also be directed, via library calls or the `perlcc` command-line script (itself written in Perl), to translate source code to either bytecode or machine code. In the former case, the output is an "executable" file beginning with `#! /usr/bin/perl` (see Sidebar 14.4 for a discussion of the `#!` convention). If invoked from the shell, this file will feed itself back into Perl 5, which will notice that the rest of the file contains bytecode instead of source, and will perform a quick reconstruction of the syntax tree, ready for interpretation.

If directed to produce machine code, `perlcc` generates a C program, which it then runs through the C compiler. The C program builds an appropriate syntax tree and passes it directly to the Perl interpreter, bypassing both the compiler and the byte-code-to-syntax-tree reconstruction. Both the bytecode and machine code back ends are considered experimental; they do not work for all programs.

17 Code Improvement

In Chapter 15 we discussed the generation, assembly, and linking of target code in the middle and back end of a compiler. The techniques we presented led to correct but highly suboptimal code: there were many redundant computations, and inefficient use of the registers, multiple functional units, and cache of a modern microprocessor. This chapter takes a look at *code improvement*: the phases of compilation devoted to generating *good* code. For the most part we will interpret "good" to mean *fast*. In a few cases we will also consider program transformations that decrease memory requirements. On occasion a real compiler may try to minimize power consumption, dollar cost of execution on a commercial cloud server, or demand for some other resource; we will not consider these issues here.

There are several possible levels of "aggressiveness" in code improvement. In a very simple compiler, or in a "nonoptimizing" run of a more sophisticated compiler, we can use a *peephole optimizer* to peruse already-generated target code for obviously suboptimal sequences of adjacent instructions. At a slightly higher level, typical of the baseline behavior of production-quality compilers, we can generate near-optimal code for *basic blocks*. As described in Chapter 15, a basic block is a maximal-length sequence of instructions that will always execute in its entirety (assuming it executes at all). In the absence of delayed branches, each basic block in assembly language or machine code begins with the target of a branch or with the instruction after a conditional branch, and ends with a branch or with the instruction before the target of a branch. As a result, in the absence of hardware exceptions, control never enters a basic block except at the beginning, and never exits except at the end. Code improvement at the level of basic blocks is known as *local* optimization. It focuses on the elimination of redundant operations (e.g., unnecessary loads or common subexpression calculations), and on effective instruction scheduling and register allocation.

At higher levels of aggressiveness, production-quality compilers employ techniques that analyze entire subroutines for further speed improvements. These techniques are known as *global* optimization.[1] They include multi-basic-block versions of redundancy elimination, instruction scheduling, and register allocation, plus code modifications designed to improve the performance of loops. Both global redundancy elimination and loop improvement typically employ a *control flow graph* representation of the program, as described in Section 15.1.1. Both employ a family of algorithms known as *data flow analysis* to trace the flow of information across the boundaries between basic blocks.

At the highest levels of aggressiveness, compilers may perform various forms of *interprocedural* code improvement. Interprocedural improvement is difficult for two main reasons. First, because a subroutine may be called from many different places in a program, it is difficult to identify (or fabricate) conditions (available registers, common subexpressions, etc.) that are guaranteed to hold at all call sites. Second, because many subroutines are separately compiled, an interprocedural code improver must generally subsume some of the work of the linker.

In the sections below we consider peephole, local, and global code improvement. We will not cover interprocedural improvement; interested readers are referred to other texts (see the Bibliographic Notes at the end of the chapter). Moreover, even for the subjects we cover, our intent will be more to "demystify" code improvement than to describe the process in detail. Much of the discussion (beginning in Section C-17.3) will revolve around the successive refinement of code for a single subroutine. This extended example will allow us to illustrate the effect of several key forms of code improvement without dwelling on the details of how they are achieved. Entire books continue to be written on code improvement; it remains a very active research topic.

As in most texts, we will sometimes refer to code improvement as "optimization," though this term is really a misnomer: we will seldom have any guarantee that our techniques will lead to optimal code. As it turns out, even some of the relatively simple aspects of code improvement (e.g., minimizing the number of registers needed in a basic block) can be shown to be NP-hard. True optimization is a realistic option only for small, special-purpose program fragments [Mas87]. Our discussion will focus on the improvement of code for imperative programs. Optimizations specific to functional or logic languages are beyond the scope of this book.

We begin in Section C-17.1 with a more detailed consideration of the phases of code improvement. We then turn to peephole optimization in Section C-17.2. It can be performed in the absence of other optimizations if desired, and the discussion introduces some useful terminology. In Sections C-17.3 and C-17.4 we consider local and global redundancy elimination. Sections C-17.5 and C-17.7 cover code improvement for loops. Section C-17.6 covers instruction scheduling. Section C-17.8 covers register allocation.

[1] The adjective "global" is standard but somewhat misleading in this context, since the improvements do not consider the program as a whole; "subroutine-level" might be more accurate.

17.1 Phases of Code Improvement

EXAMPLE 17.1

Code improvement phases

As we noted in Chapter 15, the structure of the middle and back end varies considerably from compiler to compiler. For simplicity of presentation we will continue to focus on the structure introduced in Section 15.1. In that section (as in Section 1.6) we characterized machine-independent and machine-specific code improvement as individual phases of compilation, separated by target code generation. We must now acknowledge that this was an oversimplification. In reality, code improvement is a substantially more complicated process, often comprising a very large number of phases. As noted in Section C-15.2.1, gcc has more than 140 phases in its middle end, and 70 in the back end—far more than we can cover in this chapter. In some cases optimizations depend on one another, and must be performed in a particular order. In other cases they are independent, and can be performed in any order. In still other cases it can be important to *repeat* an optimization, in order to recognize new opportunities for improvement that were not visible until some other optimization was applied.

We will concentrate in our discussion on the forms of code improvement that tend to achieve the largest increases in execution speed, and are most widely used. Compiler phases to implement these improvements are shown in Figure C-17.1. Within this structure, the middle end begins with intermediate code generation. This phase identifies fragments of the syntax tree that correspond to basic blocks. It then creates a control flow graph in which each node contains a linear sequence of three-address instructions for an idealized machine, typically one with an unlimited supply of *virtual registers*. The (machine-specific) back end begins with target code generation. This phase strings the basic blocks together into a linear program, translating each block into the instruction set of the target machine and generating branch instructions that correspond to the arcs of the control flow graph.

Machine-independent code improvement in Figure C-17.1 is shown as three key phases. The first of these identifies and eliminates redundant loads, stores, and computations within each basic block. The second deals with similar redundancies across the boundaries between basic blocks (but within the bounds of a single subroutine). The third effects several improvements specific to loops; these are particularly important, since most programs spend most of their time in loops. In Sections C-17.4, C-17.5, and C-17.7, we shall see that global redundancy elimination and loop improvement may actually be subdivided into several separate phases.

We have shown machine-specific code improvement as four separate phases. The first and third of these are essentially identical. As we noted in Section C-5.5.2, register allocation and instruction scheduling tend to interfere with one another: the instruction schedules that do the best job of minimizing pipeline stalls tend to increase the demand for architectural registers (this demand is commonly known as *register pressure*). A common strategy, assumed in our discussion, is to schedule instructions first, then allocate architectural registers, then schedule instructions again. If it turns out that there aren't enough architectural registers to go around, the register allocator will generate additional load and store instructions to *spill*

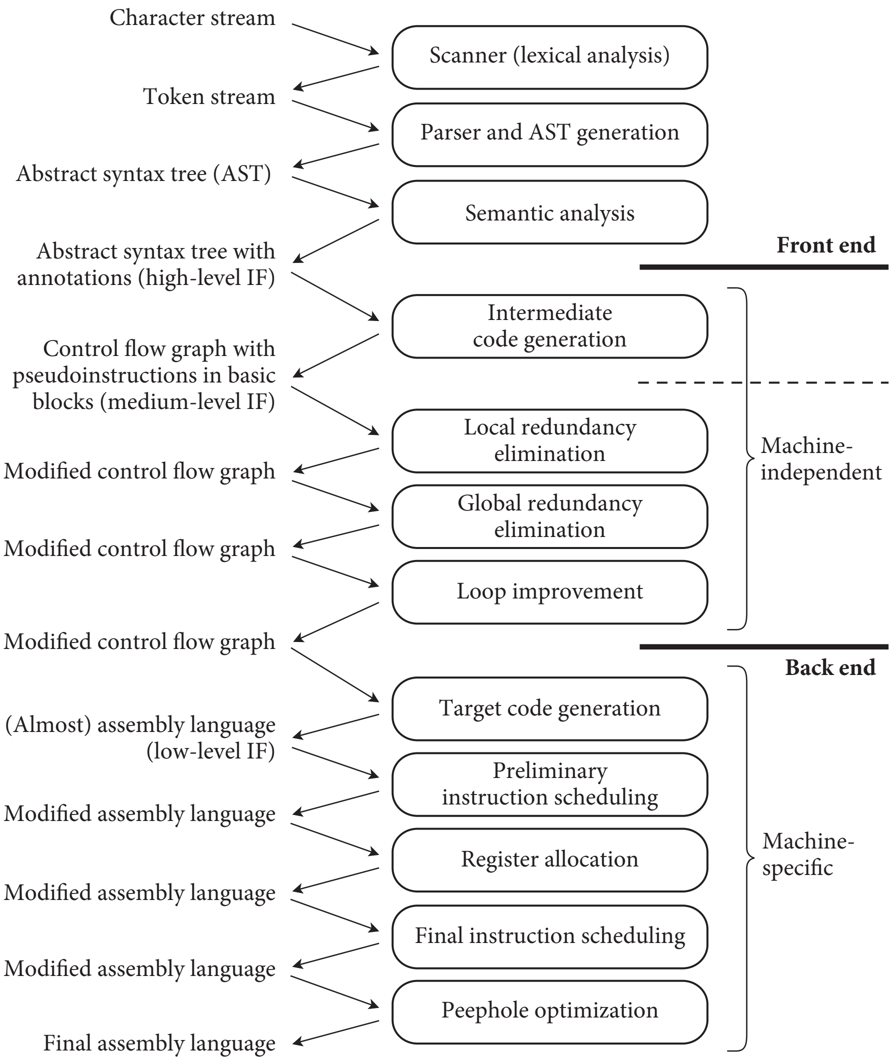

Character stream
Scanner (lexical analysis)
Token stream
Parser and AST generation
Abstract syntax tree (AST)
Semantic analysis
Abstract syntax tree with annotations (high-level IF)
Front end
Intermediate code generation
Control flow graph with pseudoinstructions in basic blocks (medium-level IF)
Local redundancy elimination
Modified control flow graph
Global redundancy elimination
Modified control flow graph
Loop improvement
Machine-independent
Modified control flow graph
Back end
Target code generation
(Almost) assembly language (low-level IF)
Preliminary instruction scheduling
Modified assembly language
Register allocation
Modified assembly language
Final instruction scheduling
Modified assembly language
Peephole optimization
Final assembly language
Machine-specific

Figure 17.1 A more detailed view of the compiler structure originally presented in Figure 15.1. Both machine-independent and machine-specific code improvement have been divided into multiple phases. As before, the dashed line shows a common "break point" for a two-pass compiler. Machine-independent code improvement may sometimes be located in a separate "middle end" pass.

registers temporarily to memory. The second round of instruction scheduling attempts to fill any delays induced by the extra loads. ■

17.2 Peephole Optimization

In a simple compiler with no machine-independent code improvement, a code generator can simply walk the abstract syntax tree, producing naive code, either as output to a file or global list, or as annotations in the tree. As we saw in Chapters 1 and 15, however, the result is generally of very poor quality (contrast the code of Example 1.2 with that of Figure 1.7). Among other things, every use of a variable as an r-value results in a load, and every assignment results in a store.

A relatively simple way to significantly improve the quality of naive code is to run a *peephole optimizer* over the target code. A peephole optimizer works by sliding a several-instruction window (a peephole) over the target code, looking for suboptimal patterns of instructions. The set of patterns to look for is heuristic; generally one creates patterns to match common suboptimal idioms produced by a particular code generator, or to exploit special instructions available on a given machine. Here are a few examples:

EXAMPLE 17.2

Elimination of redundant loads and stores: The peephole optimizer can often recognize that the value produced by a load instruction is already available in a register. For example:

```
r2 := r1 + 5
i := r2                      r2 := r1 + 5
r3 := i          becomes     i := r2
r3 := r3 × 3                 r3 := r2 × 3
```

In a similar but less common vein, if there are two stores to the same location within the optimizer's peephole (with no possible intervening load from that location), then we can generally eliminate the first. ■

EXAMPLE 17.3

Constant folding: A naive code generator may produce code that performs calculations at run time that could actually be performed at compile time. A peephole optimizer can often recognize such code. For example:

```
r2 := 3 × 2      becomes     r2 := 6
```
■

EXAMPLE 17.4

Constant propagation: Sometimes we can tell that a variable will have a constant value at a particular point in a program. We can then replace occurrences of the variable with occurrences of the constant:

```
r2 := 4                      r2 := 4                      r3 := r1 + 4
r3 := r1 + r2    becomes     r3 := r1 + 4     and then    r2 := . . .
r2 := . . .                  r2 := . . .
```

The final assignment to r2 tells us that the previous value (the 4) in r2 was *dead*—it was never going to be needed. (By analogy, a value that may be needed in some future computation is said to be *live.*) Loads of dead values can be eliminated. Similarly,

```
r2 := 4                          r3 := r1 + 4                 
r3 := r1 + r2      becomes       r3 := *r3         and then    r3 := *(r1 +4)
r3 := *r3                        r2 := . . .                   r2 := . . .
r2 := . . .
```

(This again leverages that fact that the 4 in r2 is dead at the final assignment.)

Often constant folding will reveal an opportunity for constant propagation. Sometimes the reverse occurs:

```
r1 := 3                          r1 := 3                       r1 := 3
r2 := r1 × 2       becomes       r2 := 3 × 2       and then    r2 := 6
```

If the 3 in r1 is dead, then the initial load can also be eliminated.

EXAMPLE 17.5

Common subexpression elimination: When the same calculation occurs twice within the peephole of the optimizer, we can often eliminate the second calculation:

```
r2 := r1 × 5                     r4 := r1 × 5
r2 := r2 + r3      becomes       r2 := r4 + r3
r3 := r1 × 5                     r3 := r4
```

Often, as shown here, an extra register will be needed to hold the common value.

EXAMPLE 17.6

Copy propagation: Even when we cannot tell that the contents of register *b* will be constant, we may sometimes be able to tell that register *b* will contain the same value as register *a*. We can then replace uses of *b* with uses of *a*, so long as neither *a* nor *b* is modified:

```
r2 := r1                         r2 := r1
r3 := r1 + r2      becomes       r3 := r1 + r1     and then    r3 := r1 + r1
r2 := 5                          r2 := 5                       r2 := 5
```

Performed early in code improvement, copy propagation can serve to decrease register pressure. In a peephole optimizer it may allow us (as in this case, in which the copy of r1 in r2 is dead) to eliminate one or more instructions.

EXAMPLE 17.7

Strength reduction: Numeric identities can sometimes be used to replace a comparatively expensive instruction with a cheaper one. In particular, multiplication or division by powers of two can be replaced with adds or shifts:

```
r1 := r2 × 2       becomes       r1 := r2 + r2     or          r1 := r2 << 1

r1 := r2 / 2       becomes       r1 := r2 >> 1
```

(This last replacement may not be correct when r2 is negative; see Exercise C-17.1.) In a similar vein, algebraic identities allow us to perform simplifications like the following:

```
r1 := r2 × 0       becomes       r1 := 0
```

EXAMPLE 17.8

Elimination of useless instructions: Instructions like the following can be dropped entirely:

```
r1 := r1 + 0
r1 := r1 × 1
```

■

Filling of load and branch delays: Several examples of delay-filling transformations were presented in Section C-5.5.1.

EXAMPLE 17.9

Exploitation of the instruction set: Particularly on CISC machines, sequences of simple instructions can often be replaced by a smaller number of more complex instructions. For example,

```
r1 := r1 & 0x0000FF00
r1 := r1 >> 8
```

can be replaced by an “extract byte” instruction. The sequence

```
r1 := r2 + 8
r3 := *r1
```

where r1 is dead at the end can be replaced by a single load of r3 using a base plus displacement addressing mode. Similarly,

```
r1 := *r2
r2 := r2 + 4
```

where *r2 is a 4-byte quantity can be replaced by a single load with an auto-increment addressing mode. On many machines, a series of loads from consecutive locations can be replaced by a single, multiple-register load. ■

Because they use a small, fixed-size window, peephole optimizers tend to be very fast: they impose a small, constant amount of overhead per instruction. They are also relatively easy to write and, when used on naive code, can yield dramatic performance improvements.

It should be emphasized, however, that most of the forms of code improvement in Examples C-17.2 through C-17.9 are not specific to peephole optimization. In fact, all but the last (exploitation of the instruction set) will appear in our discussion of more general forms of code improvement. The more general forms will do a better job, because they won’t be limited to looking at a narrow window of instructions. In a compiler with good machine-specific and machine-independent code improvers,

DESIGN & IMPLEMENTATION

17.1 Peephole optimization

In many cases, it is easier to count on the code improver to catch and fix suboptimal idioms than it is to generate better code in the first place. Even a peephole optimizer will catch such common examples as multiplication by one or addition of zero; there is no point adding complexity to the code generator to treat these cases specially.

there may be no need for the peephole optimizer to eliminate redundancies or useless instructions, fold constants, perform strength reduction, or fill load and branch delays. In such a compiler the peephole optimizer serves mainly to exploit idiosyncrasies of the target machine, and perhaps to clean up certain suboptimal code idioms that leak through the rest of the back end.

17.3 Redundancy Elimination in Basic Blocks

To implement local optimizations, the compiler must first identify the fragments of the syntax tree that correspond to basic blocks, as described in Section 15.1.1. Roughly speaking, these fragments consist of tree nodes that are adjacent according to in-order traversal, and contain no selection or iteration constructs. In Figure 15.6, we presented inference rules to generate linear (goto-containing) code for simple syntax trees. A similar set of inference rules can be used to create a control flow graph (Exercise 15.6).

A call to a user subroutine within a control flow graph could be treated as a pair of branches, defining a boundary between basic blocks, but as long as we know that the call will return we can simply treat it as an instruction with potentially wide-ranging side effects (i.e., as an instruction that may overwrite many registers and memory locations). As we noted in Section 9.2.4, the compiler may also choose to expand small subroutines in-line. In this case the behavior of the "call" is completely visible. If the called routine consists of a single basic block, it becomes a part of the calling block. If it consists of multiple blocks, its prologue and epilogue become part of the blocks before and after the call.

17.3.1 A Running Example

EXAMPLE 17.10
The `combinations` subroutine

Throughout much of the remainder of this chapter we will trace the improvement of code for a single subroutine: specifically, one that calculates into an array the

DESIGN & IMPLEMENTATION

17.2 Basic blocks

Many of a program's basic blocks are obvious in the source. Some, however, are created by the compiler during the translation process. Loops may be created, for example, to copy or initialize large records or subroutine parameters. Run-time semantic checks, likewise, induce large numbers of implicit selection statements. Moreover, as we shall see in Sections C-17.4.2, C-17.5, and C-17.7, many optimizations move code from one basic block to another, create or destroy basic blocks, or completely restructure loop nests. As a result of these optimizations, the final control flow graph may be very different from what the programmer might naively expect.

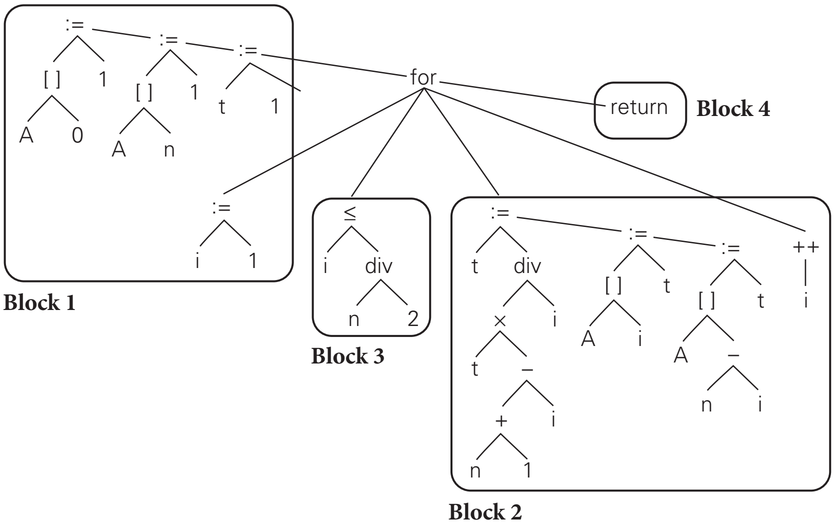


Figure 17.2 **Syntax tree for the `combinations` subroutine.** Portions of the tree corresponding to basic blocks have been circled.

binomial coefficients $\binom{n}{m}$ for all $0 \leq m \leq n$. These are the elements of the nth row of Pascal's triangle. The mth element of the row indicates the number of distinct combinations of m items that may be chosen from among a collection of n items. In C, the code looks like this:

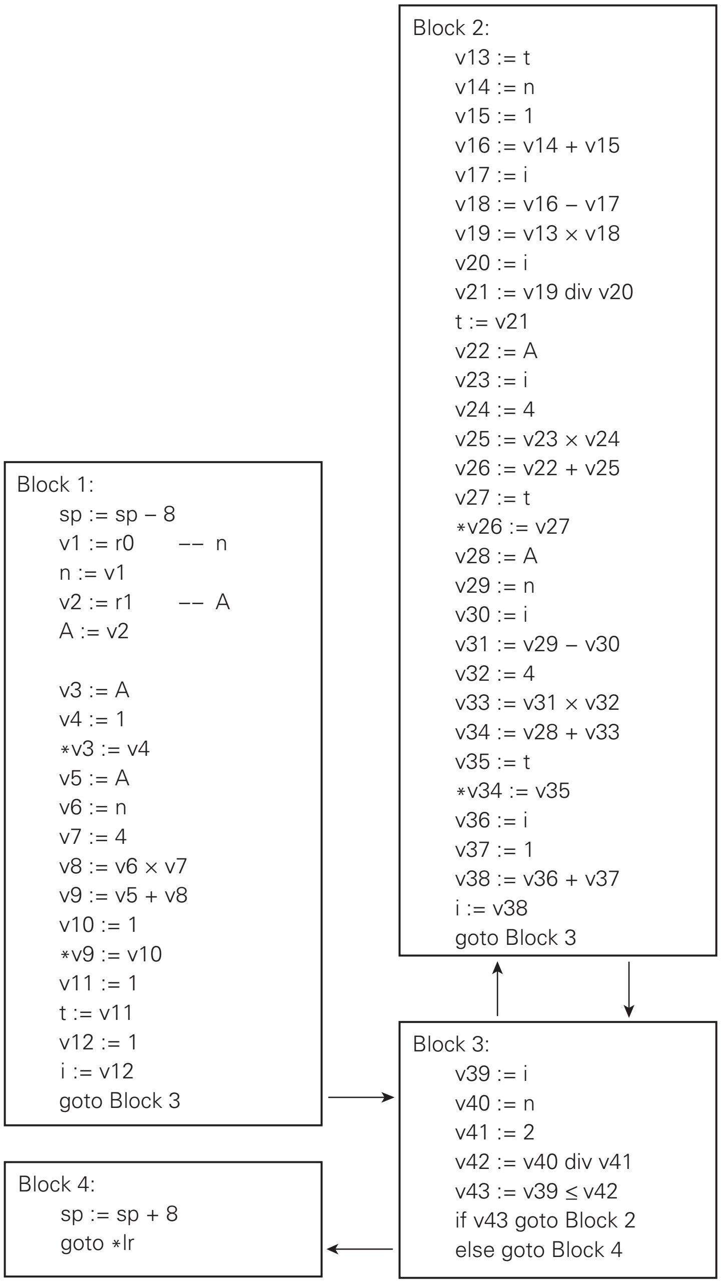
```
void combinations(int n, int *A) {
    int i, t;
    A[0] = 1;
    A[n] = 1;
    t = 1;
    for (i = 1; i <= n/2; i++) {
        t = (t * (n+1-i)) / i;
        A[i] = t;
        A[n-i] = t;
    }
}
```

This code capitalizes on the fact that $\binom{n}{m} = \binom{n}{n-m}$ for all $0 \leq m \leq n$. One can prove (Exercise C-17.2) that the use of integer arithmetic will not lead to round-off errors.

EXAMPLE 17.11

Syntax tree and naive control flow graph

A syntax tree for our subroutine appears in Figure C-17.2, with basic blocks identified. The corresponding control flow graph appears in Figure C-17.3. To avoid artificial interference between instructions at this early stage of code improvement, we employ a medium-level intermediate form (IF) in which every calculated value is placed in a separate register. To emphasize that these are virtual registers (of



```
Block 1:
      sp := sp – 8
      v1 := r0       ––  n
      n := v1
      v2 := r1       ––  A
      A := v2

      v3 := A
      v4 := 1
      *v3 := v4
      v5 := A
      v6 := n
      v7 := 4
      v8 := v6 × v7
      v9 := v5 + v8
      v10 := 1
      *v9 := v10
      v11 := 1
      t := v11
      v12 := 1
      i := v12
      goto Block 3
```

```
Block 2:
      v13 := t
      v14 := n
      v15 := 1
      v16 := v14 + v15
      v17 := i
      v18 := v16 – v17
      v19 := v13 × v18
      v20 := i
      v21 := v19 div v20
      t := v21
      v22 := A
      v23 := i
      v24 := 4
      v25 := v23 × v24
      v26 := v22 + v25
      v27 := t
      *v26 := v27
      v28 := A
      v29 := n
      v30 := i
      v31 := v29 – v30
      v32 := 4
      v33 := v31 × v32
      v34 := v28 + v33
      v35 := t
      *v34 := v35
      v36 := i
      v37 := 1
      v38 := v36 + v37
      i := v38
      goto Block 3
```

```
Block 3:
      v39 := i
      v40 := n
      v41 := 2
      v42 := v40 div v41
      v43 := v39 ≤ v42
      if v43 goto Block 2
      else goto Block 4
```

```
Block 4:
      sp := sp + 8
      goto *lr
```

Figure 17.3 Naive control flow graph for the `combinations` subroutine. Note that reference parameter A contains the *address* of the array into which to write results; hence we write v3 := A instead of v3 := &A.

which there is an unlimited supply), we name them v1, v2, ... We will use r1, r2, ... to represent architectural registers in Section C-17.8.

The fact that no virtual register is assigned a value by more than one instruction in the original control flow graph is crucial to the success of our code improvement techniques. Informally, it says that every value that could eventually end up in a separate architectural register will, at least at first, be placed in a separate virtual register. Of course if an assignment to a virtual register appears within a loop, then the register may take on a different value in every iteration. In addition, as we move through the various phases of code improvement we will relax our rules to allow a virtual register to be assigned a value in more than one place. The key point is that by employing a new virtual register whenever possible at the outset we maximize the degrees of freedom available to later phases of code improvement.

In the initial (entry) and final (exit) blocks, we have included code for the subroutine prologue and epilogue. We have assumed naive Arm calling conventions, as described in Section C-9.2.2. We have also assumed that the compiler has recognized that our subroutine is a leaf, and that it therefore has no need to save the return address (link register—lr) or frame pointer (r7) registers. In all cases, accesses to n, A, i, and t in memory should be interpreted as performing the appropriate displacement addressing with respect to the stack pointer (sp) register. Though we assume that parameter values were passed in registers (architectural registers r0 and r1 on Arm), our original (naive) code immediately saves these values to memory, so that subsequent accesses can be handled in the same way as they are for local variables. We make the saves by way of virtual registers so that they will be visible to the global value numbering algorithm described in Section C-17.4.1. Eventually, after several stages of improvement, we will find that both the parameters and the local variables can be kept permanently in registers, eliminating the need for the various loads, stores, and copy operations. ■

17.3.2 Value Numbering

To improve the code within basic blocks, we need to minimize loads and stores, and to identify redundant calculations. One common way to accomplish these tasks is to translate the syntax tree for a basic block into an *expression DAG* (directed acyclic graph) in which redundant loads and computations are merged into individual nodes with multiple parents [ALSU07, Secs. 6.1.1 and 8.5.1]. Similar functionality can also be obtained without an explicitly graphical program representation, through a technique known as local *value numbering* [Muc97, Sec. 12.4]. We describe this technique below.

Value numbering assigns the same name (a "number"—historically, a table index) to any two or more symbolically equivalent computations ("values"), so that redundant instances will be recognizable by their common name. In the formulation here, our names are virtual registers, which we merge whenever they are guaranteed to hold a common value. While performing local value numbering, we will also implement local constant folding, constant propagation, copy propagation, common subexpression elimination, strength reduction, and useless instruction elimination. (The distinctions among these optimizations will be clearer in the global case.)

We scan the instructions of a basic block in order, maintaining a dictionary to keep track of values that have already been loaded or computed, and writing instructions to a new, improved basic block that will replace the original one. For a load instruction, v*i* := x, we consult the dictionary to see whether x is already in some register v*j*. If so, we simply add an entry to the dictionary indicating that uses of v*i* should be replaced by uses of v*j*. If x is not in the dictionary, we generate a load in the new version of the basic block, and add an entry to the dictionary indicating that x is available in v*i*. For a load of a constant, v*i* := c, we check to see whether c is small enough to fit in the immediate operand of a compute instruction. If so, we add an entry to the dictionary indicating that uses of v*i* should be replaced by uses of the constant, but we generate no code: we'll embed the constant directly in the appropriate instructions when we come to them. If the constant is large, we consult the dictionary to see whether it has already been loaded (or computed) into some other register v*j*; if so, we note that uses of v*i* should be replaced by uses of v*j*. If the constant is large and not already available, then we generate instructions to load it into v*i* and then note its availability with an appropriate dictionary entry. In all cases, we create a dictionary entry for the target register of a load, indicating whether that register (1) should be used under its own name in subsequent instructions, (2) should be replaced by uses of some other register, or (3) should be replaced by some small immediate constant.

For a compute instruction, v*i* := v*j* *op* v*k*, we first consult the dictionary to see whether uses of v*j* or v*k* should be replaced by uses of some other registers or small constants v*l* and v*m*. If both operands are constants, then we can perform the operation at compile time, effecting constant folding. We then treat the constant as we did for loads above: keeping a note of its value if small, or of the register in which it resides if large. We also note opportunities to perform strength reduction or to eliminate useless instructions. If at least one of the operands is nonconstant (and the instruction is not useless), we consult the dictionary again to see whether the result of the (potentially modified) computation is already available in some register v*n*. This final lookup operation is keyed by a combination of the operator *op* and the operand registers or constants v*j* (or v*l*) and v*k* (or v*m*). If the lookup is successful, we add an entry to the dictionary indicating that uses of v*i* should be replaced by uses of v*n*. If the lookup is unsuccessful, we generate an appropriate instruction (e.g., v*i* := v*j* *op* v*k* or v*i* := v*l* *op* v*m*) in the new version of the basic block, and add a corresponding entry to the dictionary.

As we work our way through the basic block, the dictionary provides us with four kinds of information:

1. For each already-computed virtual register: whether it should be used under its own name, replaced by some other register, or replaced by an immediate constant
2. For certain variables: what register holds the (current) value

3. For certain large constants: what register holds the value
4. For some (op, arg1, arg2) triples, where arg*i* can be a register name or a constant: what register already holds the result

For a store instruction, x := v*i*, we remove any existing entry for x in the dictionary, and add an entry indicating that x is available in v*i*. We also note (in that entry) that the value of x in memory is stale. If x may be an alias for some other variable y, we must also remove any existing entry for y from the dictionary. (If we are *certain* that y is an alias for x, then we can add an entry indicating that the value of y is available in v*i*.) A similar precaution, ignored in the discussion above, applies to loads: if x may be an alias for y, and if there is an entry for y in the dictionary indicating that the value in memory is stale, then a load instruction v*i* := x must be preceded by a store to y. When we reach the end of the block, we traverse the dictionary, generating store instructions for all variables whose values in memory are stale. If any variables may be aliases for each other, we must take care to generate the stores in the order in which the values were produced. After generating the stores, we generate the branch (if any) that ends the block.

Local Code Improvement

In the process of local value numbering we automatically perform several important operations. We identify common subexpressions (none of which occur in

DESIGN & IMPLEMENTATION

17.3 Common subexpressions

It is natural to think of common subexpressions as something that could be eliminated at the source code level, and programmers are sometimes tempted to do so. The following, for example,

```
x = a + b + c;
y = a + b + d;
```

could be replaced with

```
t = a + b;
x = t + c;
y = t + d;
```

Such changes do not always make the code easier to read, however, and if the compiler is doing its job they don't make it any faster either. Moreover numerous examples of common subexpressions are entirely invisible in the source code. Examples include array subscript calculations (Section 8.2.3), references to variables in lexically enclosing scopes (Section 9.2), and references to nearby fields in complex records (Section 8.1.3). Like the pointer arithmetic discussed in Sidebar 8.8, hand elimination of common subexpressions, unless it makes the code easier to read, is usually not a good idea.

our `combinations` example), allowing us to compute them only once. We also implement constant folding and certain strength reductions. Finally, we perform local constant and copy propagation, and eliminate redundant loads and stores: our use of the dictionary to delay store instructions ensures that (in the absence of potential aliases) we never write a variable twice, or write and then read it again within the same basic block.

To increase the number of common subexpressions we can find, we may want to traverse the syntax tree prior to linearizing it, rearranging expressions into some sort of normal form. For commutative operations, for example, we can swap subtrees if necessary to put operands in lexicographic order. We can then recognize that `a + b` and `b + a` are common subexpressions. In some cases (e.g., in the context of array address calculations, or with explicit permission from the programmer), we may use associative or distributive rules to normalize expressions as well, though as we noted in Section 6.1.4 such changes can in general lead to arithmetic overflow or numerical instability. Unfortunately, straightforward normalization techniques will fail to recognize the redundancy in `a + b + c` and `a + c`; lexicographic ordering is simply a heuristic.

A naive approach to aliases is to assume that assignment to element *i* of an array may alter element *j*, for any *j*; that assignment through a pointer to an object of type *t* (in a type-safe language) may alter any variable of that type; and that a call to a subroutine may alter any variable visible in the subroutine's scope (including at a minimum all globals). These assumptions are overly conservative and can greatly limit the ability of a compiler to generate good code. More aggressive compilers perform extensive symbolic analysis of array subscripts in order to narrow the set of potential aliases for an array assignment. Similar analysis may be able to determine that particular array or record elements can be treated as unaliased scalars, making them candidates for allocation to registers. Recent years have also seen the development of very good alias analysis techniques for pointers (see Sidebar C-17.4).

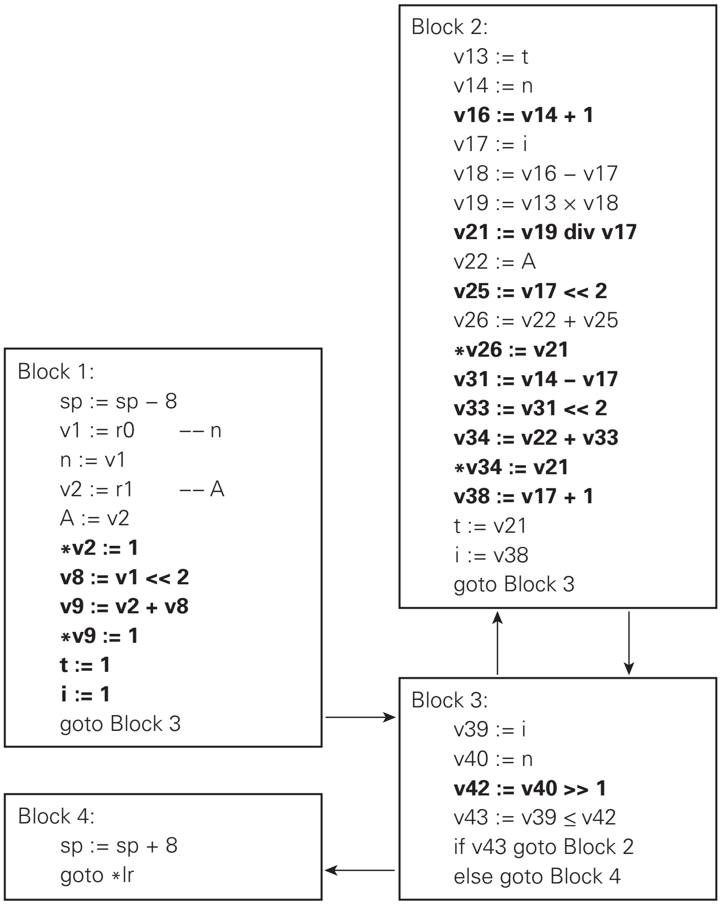
EXAMPLE 17.12
Result of local redundancy elimination

Figure C-17.4 shows the control flow graph for our `combinations` subroutine after local redundancy elimination. We have eliminated 21 of the instructions in Figure C-17.3, all of them loads of variables or constants. Thirteen of the eliminated

DESIGN & IMPLEMENTATION

17.4 Pointer analysis

The tendency of pointers to introduce aliases is one of the reasons why Fortran compilers have traditionally produced faster code than C compilers. Prior to Fortran 90, the language had no pointers, and many Fortran programs are still written without them. C programs, by contrast, tend to be pointer-rich. Some time ago, alias analysis for pointers reached the point at which good C compilers could rival their Fortran counterparts; it remains an active research topic. For a survey of the field as of 2015, see the tutorial by Smaragdakis and Balatsouras [SB15].



```
Block 1:
      sp := sp − 8
      v1 := r0      −− n
      n := v1
      v2 := r1      −− A
      A := v2
      *v2 := 1
      v8 := v1 << 2
      v9 := v2 + v8
      *v9 := 1
      t := 1
      i := 1
      goto Block 3
```

```
Block 2:
      v13 := t
      v14 := n
      v16 := v14 + 1
      v17 := i
      v18 := v16 − v17
      v19 := v13 × v18
      v21 := v19 div v17
      v22 := A
      v25 := v17 << 2
      v26 := v22 + v25
      *v26 := v21
      v31 := v14 − v17
      v33 := v31 << 2
      v34 := v22 + v33
      *v34 := v21
      v38 := v17 + 1
      t := v21
      i := v38
      goto Block 3
```

```
Block 3:
      v39 := i
      v40 := n
      v42 := v40 >> 1
      v43 := v39 ≤ v42
      if v43 goto Block 2
      else goto Block 4
```

```
Block 4:
      sp := sp + 8
      goto *lr
```

Figure 17.4 **Control flow graph for the `combinations` subroutine after local redundancy elimination and strength reduction.** Changes from Figure C-17.3 are shown in boldface type.

instructions are in the body of the loop (Blocks 2 and 3) where improvements are particularly important. We have also performed strength reduction on the two instructions that multiply a register by the constant 4 and the one that divides a register by 2, replacing them by equivalent shifts.

✓ CHECK YOUR UNDERSTANDING

1. Describe several increasing levels of “aggressiveness” in code improvement.

2. Give three examples of code improvements that must be performed in a particular order. Give two examples of code improvements that should probably be performed more than once (with other improvements in between).

3. What is *peephole optimization*? Describe at least four different ways in which a peephole optimizer might transform a program.
4. What is *constant folding*? *Constant propagation*? *Copy propagation*? *Strength reduction*?
5. What does it mean for a value in a register to be *live*?
6. What is a *control flow graph*? Why is it central to so many forms of global code improvement? How does it accommodate subroutine calls?
7. What is *value numbering*? What purpose does it serve?
8. Explain the connection between common subexpressions and expression rearrangement.
9. Why is it not practical in general for the programmer to eliminate common subexpressions at the source level?

17.4 Global Redundancy and Data Flow Analysis

In this section we will concentrate on the elimination of redundant loads and computations across the boundaries between basic blocks. We will translate the code of our basic blocks into *static single assignment* (SSA) form, which will allow us to perform global value numbering. Once value numbers have been assigned, we shall be able to perform global common subexpression elimination, constant propagation, and copy propagation. In a compiler both the translation to SSA form and the various global optimizations would be driven by data flow analysis. We will go into some of the details for global optimization (specifically, for the problems of identifying common subexpressions and useless store instructions) after a much more informal presentation of the translation to SSA form. We will also give data flow equations in Section C-17.5 for the calculation of *reaching definitions*, used (among other things) to move invariant computations out of loops.

Global redundancy elimination can be structured in such a way that it catches local redundancies as well, eliminating the need for a separate local pass. The global algorithms are easier to implement and to explain, however, if we assume that a local pass has already occurred. In particular, local redundancy elimination allows us to assume (in the absence of aliases, which we will ignore in our discussion) that no variable is read or written more than once in a basic block.

17.4.1 SSA Form and Global Value Numbering

Value numbering, as introduced in Section C-17.3, assigns a distinct virtual register name to every symbolically distinct value that is loaded or computed in a given body of code, allowing us to recognize when certain loads or computations are redundant. The first step in *global* value numbering is to distinguish among the values that may be written to a variable in different basic blocks. We accomplish this step using static single assignment (SSA) form.

Our initial translation to medium-level IF ensured that each virtual register was assigned a value by a unique instruction. This uniqueness was preserved by local value numbering. Variables, however, may be assigned in more than one basic block. Our translation to SSA form therefore begins by adding subscripts to variable names: a different one for each distinct store instruction. This convention makes it easier to identify global redundancies. It also explains the terminology: each subscripted variable in an SSA program has a single static (compile time) assignment—a single store instruction.

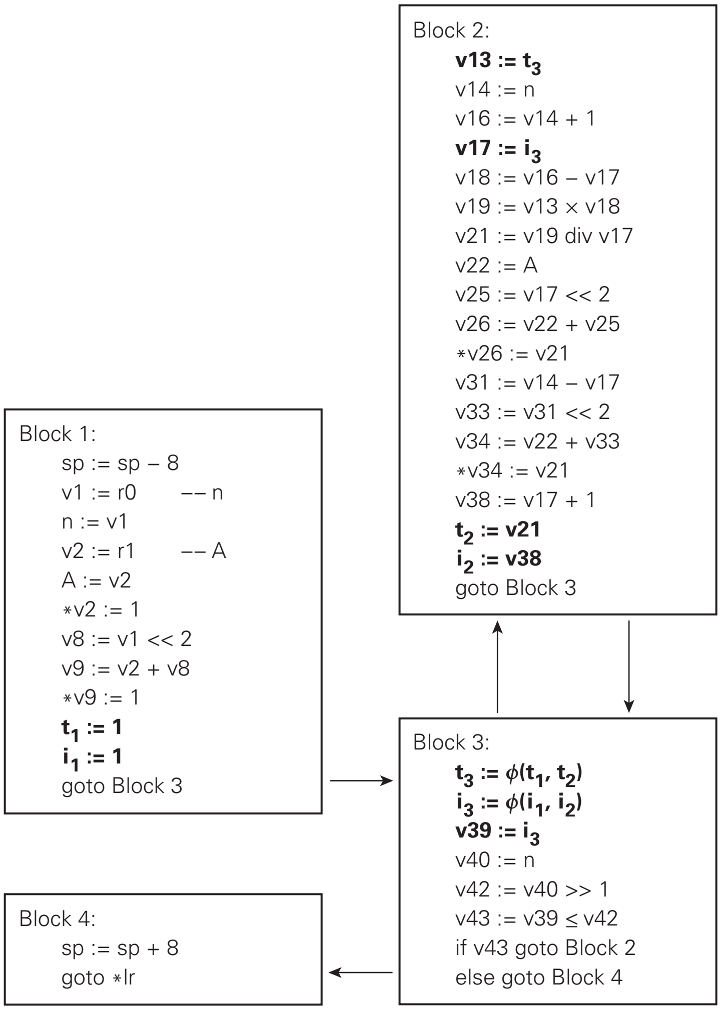
Following the flow of the program, we assign subscripts to variables in load instructions, to match the corresponding stores. If the instruction `v2 := x` is guaranteed to read the value of `x` written by the instruction x_3 := `v1`, then we replace `v2 := x` with `v2 :=` x_3. If we cannot tell which version of `x` will be read, we use a hypothetical *merge function* (also known as a *selection function*, and traditionally represented by the Greek letter ϕ) to choose among the possible alternatives. Fortunately, we won't actually have to compute merge functions at run time. Their only purpose is to help us identify possible code improvements; we will drop them (and the subscripts) prior to target code generation.

In general, the translation to SSA form (and the identification of merge functions in particular) requires the use of data flow analysis. We will describe the concept of data flow in the context of global common subexpression elimination in Section C-17.4.2. In the current subsection we will generate SSA code informally; data flow formulations can be found in more advanced compiler texts [CT11, Sec. 9.3; AK02, Sec. 4.4.4; App97, Sec. 19.1; Muc97, Sec. 8.11].

EXAMPLE 17.13
Conversion to SSA form

In the `combinations` subroutine (Figure C-17.4) we assign the subscript 1 to the stores of `t` and `i` at the end of Block 1. We assign the subscript 2 to the stores of `t` and `i` at the end of Block 2. Thus at the end of Block 1 t_1 and i_1 are live; at the end of Block 2 t_2 and i_2 are live. What about Block 3? If control enters Block 3 from Block 1, then t_1 and i_1 will be live, but if control enters Block 3 from Block 2, then t_2 and i_2 will be live. We invent a merge function ϕ that returns its first argument if control enters Block 3 from Block 1, and its second argument if control enters Block 3 from Block 2. We then use this function to write values to new names t_3 and i_3. Since Block 3 does not modify either `t` or `i`, we know that t_3 and i_3 will be live at the end of the block. Moreover, since control always enters Block 2 from Block 3, t_3 and i_3 will be live at the beginning of Block 2. The load of `v13` in Block 2 is guaranteed to return t_3; the loads of `v17` in Block 2 and of `v39` in Block 3 are guaranteed to return i_3.

SSA form annotates the right-hand sides of loads with subscripts and merge functions in such a way that at any given point in the program, if `v`i and `v`j were given values by load instructions with symbolically identical right-hand sides, then the loaded values are guaranteed to have been produced by (the same execution of) the same prior store instruction. Because ours is a simple subroutine, only one merge function is needed: it indicates whether control entered Block 3 from Block 1



```
Block 1:
    sp := sp − 8
    v1 := r0       −− n
    n := v1
    v2 := r1       −− A
    A := v2
    *v2 := 1
    v8 := v1 << 2
    v9 := v2 + v8
    *v9 := 1
    t1 := 1
    i1 := 1
    goto Block 3
```

```
Block 2:
    v13 := t3
    v14 := n
    v16 := v14 + 1
    v17 := i3
    v18 := v16 − v17
    v19 := v13 × v18
    v21 := v19 div v17
    v22 := A
    v25 := v17 << 2
    v26 := v22 + v25
    *v26 := v21
    v31 := v14 − v17
    v33 := v31 << 2
    v34 := v22 + v33
    *v34 := v21
    v38 := v17 + 1
    t2 := v21
    i2 := v38
    goto Block 3
```

```
Block 3:
    t3 := ϕ(t1, t2)
    i3 := ϕ(i1, i2)
    v39 := i3
    v40 := n
    v42 := v40 >> 1
    v43 := v39 ≤ v42
    if v43 goto Block 2
    else goto Block 4
```

```
Block 4:
    sp := sp + 8
    goto *lr
```

Figure 17.5 Control flow graph for the `combinations` subroutine, in static single assignment (SSA) form. Changes from Figure C-17.4 are shown in boldface type.

or from Block 2. In a more complicated subroutine there could be additional merge functions, for other blocks with more than one predecessor. SSA form for the `combinations` subroutine appears in Figure C-17.5. ■

EXAMPLE 17.14
Global value numbering

With flow-dependent values determined by merge functions, we are now in a position to perform global value numbering. As in local value numbering, the goal is to merge any virtual registers that are guaranteed to hold symbolically equivalent expressions.

In the local case we were able to perform a linear pass over the code, keeping a dictionary that mapped loaded and computed expressions to the names of virtual registers that contained them. This approach does not suffice in the global case, because the code may have cycles. The general solution can be formulated using data flow, or obtained with a simpler algorithm [Muc97, Sec. 12.4.2] that begins by unifying all expressions with the same top-level operator, and then repeatedly separates expressions whose operands are distinct, in a manner reminiscent of the DFA minimization algorithm of Section 2.2.1. In contrast to our presentation of local value numbering, where we performed code improvements such as eliminating redundant loads and stores, we perform only global value numbering here, leaving further code improvements to separate dataflow analyses that build on our results. Again, we perform the analysis for our running example informally.

We can begin by adopting the results of local value numbering for Block 1; since this is the first basic block and local redundancies have been removed, its virtual register names have already been merged as much as possible. In Block 2, the second instruction loads n into v14. Since we already used v1 for n in Block 1, we can substitute the same name here. This substitution violates, for the first time, our assumption that every virtual register is given a value by a single static instruction. The "violation" is safe, however: both occurrences of n have the same subscript (none at all, in this case), so we know that at any given point in the code, if v1 and v14 have both been given values, then those values are the same. We can't (yet) eliminate the load in Block 2, because we don't (yet) know that Block 1 will have executed first. For consistency we replace v14 with v1 in the third instruction of Block 2. Then, by similar reasoning, we replace v22 with v2 in the 8th, 10th, and 14th instructions.

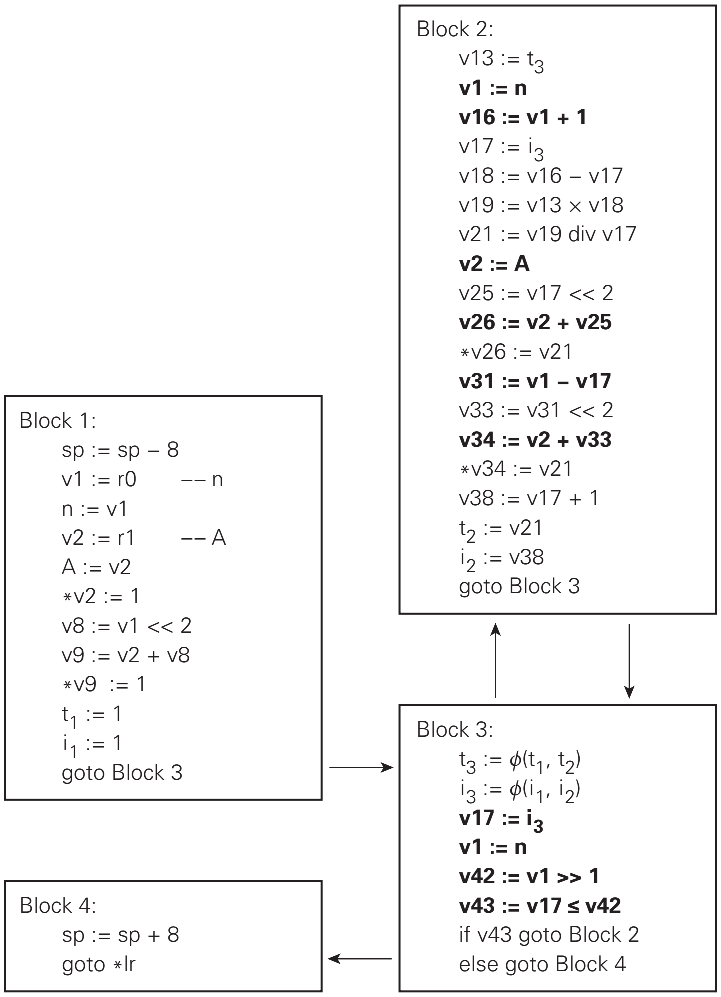
In Block 3 we have more replacements. In the first real instruction (v39 := i_3), we recall that the same right-hand side is loaded into v17 in Block 2. We therefore replace v39 with v17, in both the first and fourth instructions. Similarly, we replace v40 with v1, in both the second and third instructions. There are no changes in Block 4.

The result of global value numbering on our combinations subroutine appears in Figure C-17.6. In this case the only common values identified were variables loaded from memory. In a more complicated subroutine, we would also identify known-to-be-identical computations performed in more than one block (though we would not yet know which, if any, were redundant). As we did with loads, we would rename left-hand sides so that all symbolically equivalent computations place their results in the same virtual register.

Static single assignment form is useful for a variety of code improvements. In our discussion here we use it only for global value numbering. We will drop it in later figures.

17.4.2 Global Common Subexpression Elimination

We have seen an informal example of data flow analysis in the construction of static single assignment form. We will now employ a more formal example for global common subexpression elimination. As a result of global value numbering, we



```
Block 1:
      sp := sp − 8
      v1 := r0       −− n
      n := v1
      v2 := r1       −− A
      A := v2
      *v2 := 1
      v8 := v1 << 2
      v9 := v2 + v8
      *v9  := 1
      t1 := 1
      i1 := 1
      goto Block 3
```

```
Block 2:
      v13 := t3
      v1 := n
      v16 := v1 + 1
      v17 := i3
      v18 := v16 − v17
      v19 := v13 × v18
      v21 := v19 div v17
      v2 := A
      v25 := v17 << 2
      v26 := v2 + v25
      *v26 := v21
      v31 := v1 − v17
      v33 := v31 << 2
      v34 := v2 + v33
      *v34 := v21
      v38 := v17 + 1
      t2 := v21
      i2 := v38
      goto Block 3
```

```
Block 3:
      t3 := φ(t1, t2)
      i3 := φ(i1, i2)
      v17 := i3
      v1 := n
      v42 := v1 >> 1
      v43 := v17 ≤ v42
      if v43 goto Block 2
      else goto Block 4
```

```
Block 4:
      sp := sp + 8
      goto *lr
```

Figure 17.6 Control flow graph for the `combinations` subroutine after global value numbering. Changes from Figure C-17.5 are shown in boldface type.

know that any common subexpression will have been placed into the same virtual register wherever it is computed. We will therefore use virtual register names to represent expressions in the discussion below.[2] The goal of global common subexpression elimination is to identify places in which an instruction that computes

[2] As presented here, there is a one–one correspondence among SSA names, global value numbers, and (after global value numbering has been completed) virtual register names. Other texts and papers sometimes distinguish among these concepts more carefully, and use them for different purposes.

a value for a given virtual register can be eliminated, because the computation is certain to already have occurred on every control path leading to the instruction.

Many instances of data flow analysis can be cast in the following framework: (1) four sets for each basic block B, called In_B, Out_B, Gen_B, and $Kill_B$; (2) values for the *Gen* and *Kill* sets; (3) an equation relating the sets for any given block B; (4) an equation relating the *Out* set of a given block to the *In* sets of its successors, or relating the *In* set of the block to the *Out* sets of its predecessors; and (often) (5) certain initial conditions. The goal of the analysis is to find a *fixed point* of the equations: a consistent set of *In* and *Out* sets that satisfy both the equations and the initial conditions. Some problems have a single fixed point. Others may have more than one, in which case we usually want either the least or the greatest fixed point (smallest or largest sets).

EXAMPLE 17.15

Data flow equations for available expressions

In the case of global common subexpression elimination, In_B is the set of expressions (virtual registers) guaranteed to be available at the beginning of block B. These *available expressions* will all have been set by predecessor blocks. Out_B is the set of expressions guaranteed to be available at the end of B. $Kill_B$ is the set of expressions *killed* in B: invalidated by assignment to one of the variables used to calculate the expression, and not subsequently recalculated in B. Gen_B is the set of expressions calculated in B and not subsequently killed in B. The data flow equations for available expression analysis are[3]

$$\begin{aligned} Out_B &= Gen_B \cup (In_B \smallsetminus Kill_B) \\ In_B &= \bigcap_{\text{predecessors } A \text{ of } B} Out_A \end{aligned}$$

Our initial condition is $In_1 = \varnothing$: no expressions are available at the beginning of execution.

Available expression analysis is known as a *forward* data flow problem, because information flows forward across branches: the *In* set of a block depends on the *Out* sets of its predecessors. We shall see an example of a *backward* data flow problem later in this section. ■

EXAMPLE 17.16

Fixed point for available expressions

We calculate the desired fixed point of our equations in an inductive (iterative) fashion, much as we computed FIRST and FOLLOW sets in Section 2.3.3. Our equation for In_B uses intersection to insist that an expression be available on all paths into B. In our iterative algorithm, this means that In_B can only shrink with subsequent iterations. Because we want to find as many available expressions as possible, we therefore optimistically assume that all expressions are initially available as inputs to all blocks other than the first; that is, $In_{B,B\neq 1} = \{$n, A, t, i, v1, v2, v8, v9, v13, v16, v17, v18, v19, v21, v25, v26, v31, v33, v34, v38, v42, v43$\}$.

Our *Gen* and *Kill* sets can be found in a single backward pass over each of the basic blocks. In Block 3, for example, the last assignment defines a value for v43.

[3] Set notation here is standard: $\bigcup_i S_i$ indicates the union of all sets S_i; $\bigcap_i S_i$ indicates the intersection of all sets S_i; $A \smallsetminus B$, pronounced "A minus B" indicates the set of all elements found in A but not in B.

We therefore know that v43 is in Gen_3. Working backward, so are v42, v1, and v17. As we notice each of these, we also consider their impact on $Kill_3$. Virtual register v43 does not appear on the right-hand side of any assignment in the program (it is not part of the expression named by any virtual register), so giving it a value kills nothing. Virtual register v42 is part of the expression named by v43, but since v43 is given a value later in the block (is already in Gen_3), the assignment to v42 does not force v43 into $Kill_3$. Virtual register v1 is a different story. It is part of the expressions named by v8, v16, v31, and v42. Since v42 is already in Gen_3, we do not add it to $Kill_3$. We do, however, put v8, v16, and v31 in $Kill_3$. In a similar manner, the assignment to v17 forces v18, v21, v25, and v38 into $Kill_3$. Note that we do not have to worry about virtual registers that depend in turn on v8, v16, v18, v21, v25, v31, or v38: our iterative data flow algorithm will take care of that; all we need now is one level of dependence. Stores to program variables (e.g., at the ends of Blocks 1 and 2) kill the corresponding virtual registers.

After completing a backward scan of all four blocks, we have the following *Gen* and *Kill* sets:

$$
\begin{array}{ll}
Gen_1 = \{\mathtt{v1}, \mathtt{v2}, \mathtt{v8}, \mathtt{v9}\} & Kill_1 = \{\mathtt{v13}, \mathtt{v16}, \mathtt{v17}, \mathtt{v26}, \mathtt{v31}, \mathtt{v34}, \mathtt{v42}\} \\
Gen_2 = \{\mathtt{v1}, \mathtt{v2}, \mathtt{v13}, \mathtt{v16}, \mathtt{v17}, \mathtt{v18}, \mathtt{v19}, & Kill_2 = \{\mathtt{v8}, \mathtt{v9}, \mathtt{v13}, \mathtt{v17}, \mathtt{v42}, \mathtt{v43}\} \\
\qquad \mathtt{v21}, \mathtt{v25}, \mathtt{v26}, \mathtt{v31}, \mathtt{v33}, \mathtt{v34}, \mathtt{v38}\} & \\
Gen_3 = \{\mathtt{v1}, \mathtt{v17}, \mathtt{v42}, \mathtt{v43}\} & Kill_3 = \{\mathtt{v8}, \mathtt{v16}, \mathtt{v18}, \mathtt{v21}, \mathtt{v25}, \mathtt{v31}, \mathtt{v38}\} \\
Gen_4 = \varnothing & Kill_4 = \varnothing
\end{array}
$$

Applying the first of our data flow equations ($Out_B = Gen_B \cup (In_B \smallsetminus Kill_B)$) to all blocks, we obtain

$$
\begin{array}{l}
Out_1 = \{\mathtt{v1}, \mathtt{v2}, \mathtt{v8}, \mathtt{v9}\} \\
Out_2 = \{\mathtt{v1}, \mathtt{v2}, \mathtt{v13}, \mathtt{v16}, \mathtt{v17}, \mathtt{v18}, \mathtt{v19}, \mathtt{v21}, \mathtt{v25}, \mathtt{v26}, \mathtt{v31}, \mathtt{v33}, \mathtt{v34}, \mathtt{v38}\} \\
Out_3 = \{\mathtt{v1}, \mathtt{v2}, \mathtt{v9}, \mathtt{v13}, \mathtt{v17}, \mathtt{v19}, \mathtt{v26}, \mathtt{v33}, \mathtt{v34}, \mathtt{v42}, \mathtt{v43}\} \\
Out_4 = \{\mathtt{v1}, \mathtt{v2}, \mathtt{v8}, \mathtt{v9}, \mathtt{v13}, \mathtt{v16}, \mathtt{v17}, \mathtt{v18}, \mathtt{v19}, \mathtt{v21}, \mathtt{v25}, \mathtt{v26}, \mathtt{v31}, \mathtt{v33}, \mathtt{v34}, \mathtt{v38}, \mathtt{v42}, \mathtt{v43}\}
\end{array}
$$

If we now apply our second equation ($In_B = \bigcap_A Out_A$) to all blocks, followed by a second iteration of the first equation, we obtain

$$
\begin{array}{ll}
In_1 = \varnothing & Out_1 = \{\mathtt{v1}, \mathtt{v2}, \mathtt{v8}, \mathtt{v9}\} \\
In_2 = \{\mathtt{v1}, \mathtt{v2}, \mathtt{v9}, \mathtt{v13}, \mathtt{v17}, \mathtt{v19}, & Out_2 = \{\mathtt{v1}, \mathtt{v2}, \mathtt{v13}, \mathtt{v16}, \mathtt{v17}, \mathtt{v18}, \mathtt{v19}, \\
\qquad \mathtt{v26}, \mathtt{v33}, \mathtt{v34}, \mathtt{v42}, \mathtt{v43}\} & \qquad \mathtt{v21}, \mathtt{v25}, \mathtt{v26}, \mathtt{v31}, \mathtt{v33}, \mathtt{v34}, \mathtt{v38}\} \\
In_3 = \{\mathtt{v1}, \mathtt{v2}\} & Out_3 = \{\mathtt{v1}, \mathtt{v2}, \mathtt{v17}, \mathtt{v42}, \mathtt{v43}\} \\
In_4 = \{\mathtt{v1}, \mathtt{v2}, \mathtt{v9}, \mathtt{v13}, \mathtt{v17}, \mathtt{v19}, & Out_4 = \{\mathtt{v1}, \mathtt{v2}, \mathtt{v9}, \mathtt{v13}, \mathtt{v17}, \mathtt{v19}, \\
\qquad \mathtt{v26}, \mathtt{v33}, \mathtt{v34}, \mathtt{v42}, \mathtt{v43}\} & \qquad \mathtt{v26}, \mathtt{v33}, \mathtt{v34}, \mathtt{v42}, \mathtt{v43}\}
\end{array}
$$

One more iteration of each equation yields the fixed point:

$$
\begin{aligned}
&\mathit{In}_1 = \varnothing && \mathit{Out}_1 = \{\texttt{v1}, \texttt{v2}, \texttt{v8}, \texttt{v9}\}\\
&\mathit{In}_2 = \{\texttt{v1}, \texttt{v2}, \texttt{v17}, \texttt{v42}, \texttt{v43}\} && \mathit{Out}_2 = \{\texttt{v1}, \texttt{v2}, \texttt{v13}, \texttt{v16}, \texttt{v17}, \texttt{v18}, \texttt{v19},\\
& && \qquad \texttt{v21}, \texttt{v25}, \texttt{v26}, \texttt{v31}, \texttt{v33}, \texttt{v34}, \texttt{v38}\}\\
&\mathit{In}_3 = \{\texttt{v1}, \texttt{v2}\} && \mathit{Out}_3 = \{\texttt{v1}, \texttt{v2}, \texttt{v17}, \texttt{v42}, \texttt{v43}\}\\
&\mathit{In}_4 = \{\texttt{v1}, \texttt{v2}, \texttt{v17}, \texttt{v42}, \texttt{v43}\} && \mathit{Out}_4 = \{\texttt{v1}, \texttt{v2}, \texttt{v17}, \texttt{v42}, \texttt{v43}\}
\end{aligned}
$$

EXAMPLE 17.17

Result of global common subexpression elimination

We can now exploit what we have learned. Whenever a virtual register is in the *In* set of a block, we can drop any assignment of that register in the block. In our example subroutine, we can drop the loads of v1, v2, and v17 in Block 2, and the load of v1 in Block 3. In addition, whenever a virtual register corresponding to a variable is in the *In* set of a block, we can replace a load of that variable with a register–register move on each of the potential paths into the block. In our example, we can replace the load of t in Block 2 and the load of i in Block 3 (the load of i in Block 2 has already been eliminated). To compensate, we must load v13 and v17 with the constant 1 at the end of Block 1, and move v21 into v13 and v38 into v17 at the end of Block 2. The final result appears in Figure C-17.7.

(The careful reader may note that v21 and v38 are not strictly necessary: if we computed new values directly into v13 and v17, we could eliminate the two register–register moves. This observation, while correct, need not be made at this time; it can wait until we perform induction variable optimizations and register allocation, to be described in Sections C-17.5.2 and C-17.8, respectively.)

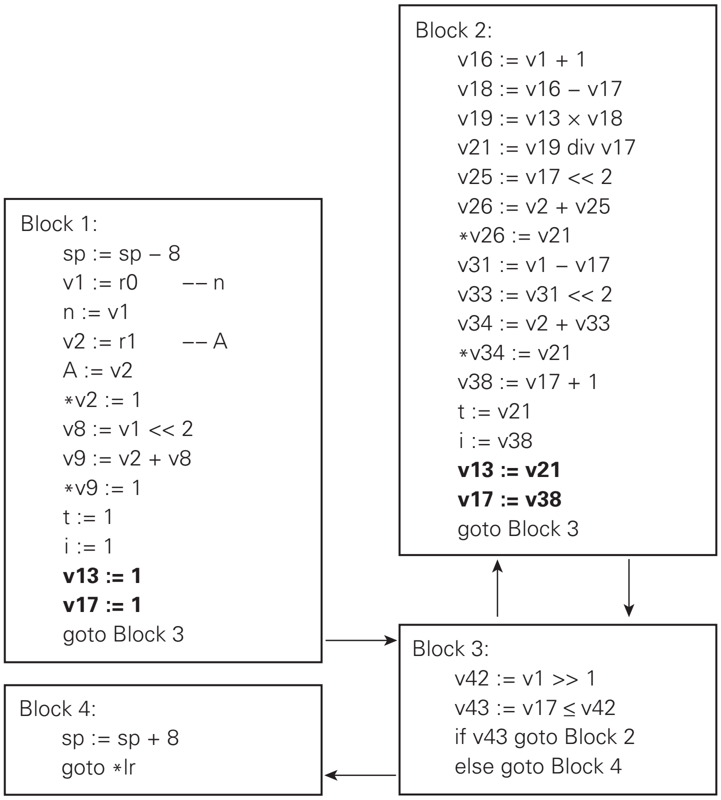
Splitting Control Flow Edges

EXAMPLE 17.18

Edge splitting transformations

If the block (call it A) in which a variable is written has more than one successor, only one of which (call it B) contains a redundant load, and if B has more than one predecessor, then we need to create a new block on the arc between A and B to hold the register–register move. This way the move will not be executed on code paths that don't need it. In a similar vein, if an expression is available from A but not from B's other predecessor, then we can move the load or computation of the expression back into the predecessor that lacks it or, if that predecessor has more than one successor, into a new block on the connecting arc. This move will eliminate a redundancy on the path through A. These "edge splitting" transformations are illustrated in Figure C-17.8. In general, a load or computation is said to be *partially redundant* if it is a repetition of an earlier load or store on some paths through the flow graph, but not on others. No edge splits are required in the `combinations` example.

Common subexpression elimination can have a complicated effect on register pressure. If we realize that the expression v10 + v20 has been calculated into, say, register v30 earlier in the program, and we exploit this knowledge to replace a later recalculation of the expression with a direct use of v30, then we may expand v30's *live range*—the span of instructions over which its value is needed. At the same time, if v10 and v20 are not used for other purposes in the intervening region of the program, we may *shrink* the range over which *they* are live. In a subroutine with a high level of register pressure, a good compiler may sometimes perform the



```
Block 1:
      sp := sp – 8
      v1 := r0      –– n
      n := v1
      v2 := r1      –– A
      A := v2
      *v2 := 1
      v8 := v1 << 2
      v9 := v2 + v8
      *v9 := 1
      t := 1
      i := 1
      v13 := 1
      v17 := 1
      goto Block 3

Block 2:
      v16 := v1 + 1
      v18 := v16 – v17
      v19 := v13 × v18
      v21 := v19 div v17
      v25 := v17 << 2
      v26 := v2 + v25
      *v26 := v21
      v31 := v1 – v17
      v33 := v31 << 2
      v34 := v2 + v33
      *v34 := v21
      v38 := v17 + 1
      t := v21
      i := v38
      v13 := v21
      v17 := v38
      goto Block 3

Block 3:
      v42 := v1 >> 1
      v43 := v17 ≤ v42
      if v43 goto Block 2
      else goto Block 4

Block 4:
      sp := sp + 8
      goto *lr
```

Figure 17.7 **Control flow graph for the `combinations` subroutine after performing global common subexpression elimination.** Note the absence of the many load instructions of Figure C-17.6. Compensating register–register moves are shown in boldface type.

inverse of common subexpression elimination (known as *forward substitution*) in order to shrink live ranges.

Live Variable Analysis

Constant propagation and copy propagation, like common subexpression elimination, can be formulated as instances of data flow analysis. We skip these analyses here; none of them yields improvements in our example. Instead, we turn our attention to *live variable analysis*, which is very important in our example, and in general in any subroutine in which global common subexpression analysis has eliminated load instructions.

Live variable analysis is the *backward* flow problem mentioned above. It determines which instructions produce values that will be needed in the future, allowing us to eliminate *dead* (useless) instructions. In our example we will concern ourselves only with values written to memory and with the elimination of dead stores. When applied to values in virtual registers as well, live variable analysis

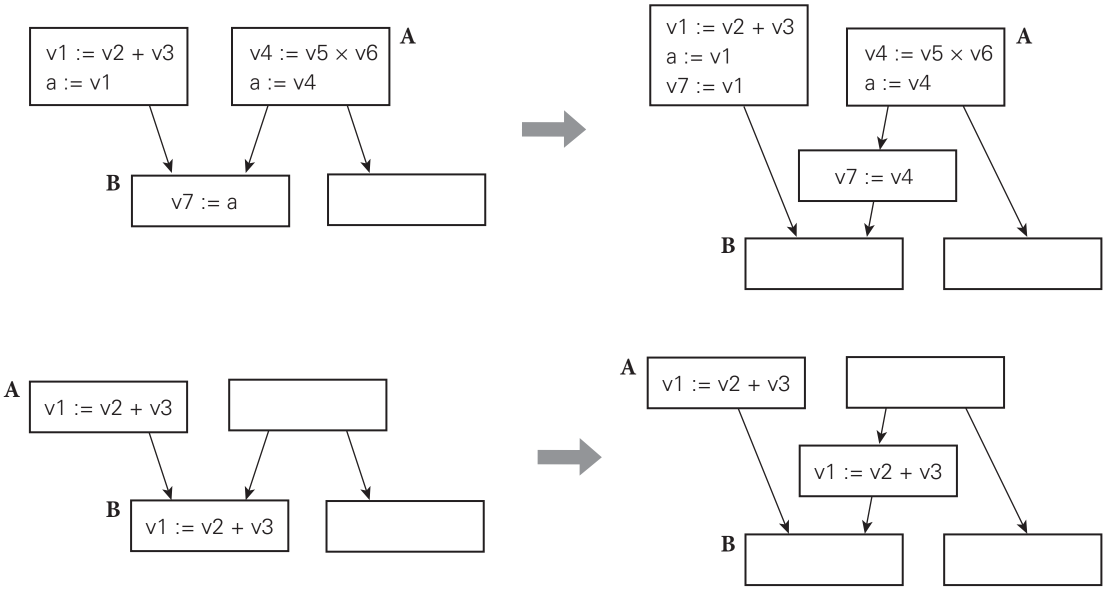


Figure 17.8 Splitting an edge of a control flow graph to eliminate a redundant load (*top*) or a partially redundant computation (*bottom*).

can help to identify other dead instructions. (None of these arise this early in the `combinations` example.)

EXAMPLE 17.19

Data flow equations for live variables

For this instance of data flow analysis, In_B is the set of variables that are live at the beginning of block B. Out_B is the set of variables that are live at the end of the block. Gen_B is the set of variables read in B without first being written in B. $Kill_B$ is the set of variables written in B without having been read first. The data flow equations are

$$\begin{aligned} In_B &= Gen_B \cup (Out_B \smallsetminus Kill_B) \\ Out_B &= \bigcup_{\text{successors } C \text{ of } B} In_C \end{aligned}$$

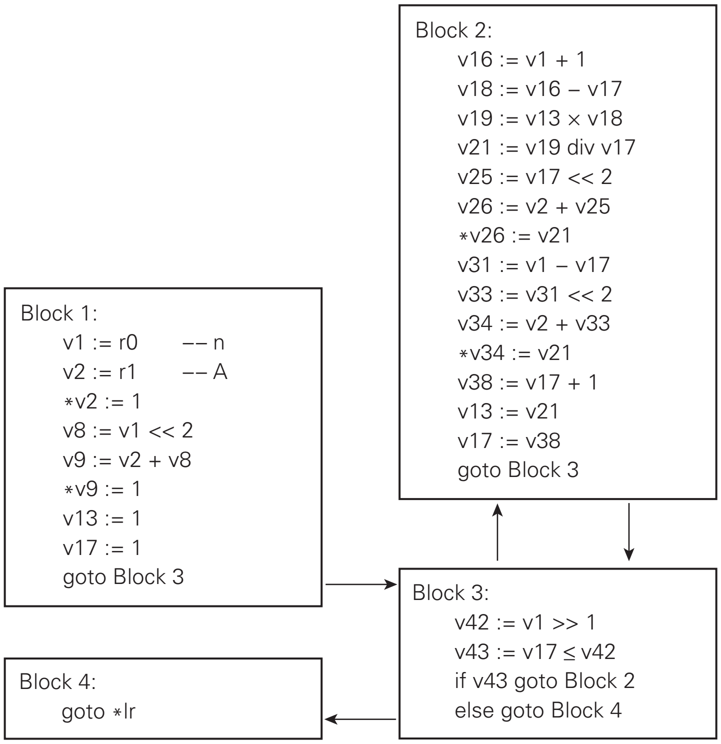
Our initial condition is $Out_4 = \varnothing$: no variables are live at the end of execution. (If our subroutine wrote any nonlocal [e.g., global] variables, these would be initial members of Out_4.)

In comparison to the equations for available expression analysis, the roles of *In* and *Out* have been reversed (that's why it's a backward problem), and the intersection operator in the second equation has been replaced by a union. Intersection ("all paths") problems require that information flow over *all* paths between blocks; union ("any path") problems require that it flow along *some* path. Further data flow examples appear in Exercises C-17.7 and C-17.9. ■

EXAMPLE 17.20

Fixed point for live variables

In our example program, we have



```
Block 1:
    v1 := r0      -- n
    v2 := r1      -- A
    *v2 := 1
    v8 := v1 << 2
    v9 := v2 + v8
    *v9 := 1
    v13 := 1
    v17 := 1
    goto Block 3
```

```
Block 2:
    v16 := v1 + 1
    v18 := v16 − v17
    v19 := v13 × v18
    v21 := v19 div v17
    v25 := v17 << 2
    v26 := v2 + v25
    *v26 := v21
    v31 := v1 − v17
    v33 := v31 << 2
    v34 := v2 + v33
    *v34 := v21
    v38 := v17 + 1
    v13 := v21
    v17 := v38
    goto Block 3
```

```
Block 3:
    v42 := v1 >> 1
    v43 := v17 ≤ v42
    if v43 goto Block 2
    else goto Block 4
```

```
Block 4:
    goto *lr
```

Figure 17.9 Control flow graph for the `combinations` subroutine after performing live variable analysis. Starting with Figure C-17.7, the compiler has eliminated all stores to n, A, t, and i. It has also dropped the changes to the stack pointer that used to appear in the subroutine prologue and epilogue: we don't need space for local variables anymore.

$$\begin{array}{llll}
Gen_1 = \varnothing & & Kill_1 = \{\mathrm{n, A, t, i}\} \\
Gen_2 = \varnothing & & Kill_2 = \{\mathrm{t, i}\} \\
Gen_3 = \varnothing & & Kill_3 = \varnothing \\
Gen_4 = \varnothing & & Kill_4 = \varnothing
\end{array}$$

Our use of union means that *Out* sets can only grow with each iteration, so we begin with $Out_B = \varnothing$ for all blocks B (not just B_4). One iteration of our data flow equations gives us $In_B = Gen_B$ and $Out_B = \varnothing$ for all blocks B. But since $Gen_B = \varnothing$ for all B, this is our fixed point! Common subexpression elimination has left us with a situation in which none of our parameters or local variables is live; all of the stores of A, n, t, and i can be eliminated. Moreover, now that computation works entirely in registers, we don't even need a stack frame: we can eliminate the updates of the stack pointer in the subroutine prologue and epilogue, leaving us with the code in Figure C-17.9. ■

Aliases must be treated in a conservative fashion in both common subexpression elimination and live variable analysis. If a store instruction might modify variable

x, then for purposes of common subexpression elimination we must consider the store as killing any expression that depends on x. If a load instruction might access x, and x is not written earlier in the block containing the load, then x must be considered live at the beginning of the block. In our example we have assumed that the compiler is able to verify that, as a reference parameter, array A cannot alias either value parameter n or local variables t and i.

✓ **CHECK YOUR UNDERSTANDING**

10. What is *static single assignment (SSA) form*? Why is SSA form needed for global value numbering, but not for local value numbering?
11. What are *merge functions* in the context of SSA form?
12. Give three distinct examples of *data flow analysis*. Explain the difference between *forward* and *backward* flow. Explain the difference between *all-paths* and *any-path* flow.
13. Explain the role of the *In*, *Out*, *Gen*, and *Kill* sets common to many examples of data flow analysis.
14. What is a *partially redundant* computation? Why might an algorithm to eliminate partial redundancies need to *split* an edge in a control flow graph?
15. What is an *available expression*?
16. What is *forward substitution*?
17. What is *live variable analysis*? What purpose does it serve?
18. Describe at least three instances in which code improvement algorithms must consider the possibility of aliases.

17.5 Loop Improvement I

Because programs tend to spend most of their time in loops, code improvements that improve the speed of loops are particularly important. In this section we consider two classes of loop improvements: those that move *invariant* computations out of the body of a loop and into its header, and those that reduce the amount of time spent maintaining *induction variables*. In Section C-17.7 we will consider transformations that improve instruction scheduling by restructuring a loop body to include portions of more than one iteration of the original loop, and that manipulate multiply nested loops to improve cache performance or increase opportunities for parallelization.

17.5.1 Loop Invariants

A *loop invariant* is an instruction (i.e., a load or calculation) in a loop whose result is guaranteed to be the same in every iteration.[4] If a loop is executed n times and we are able to move an invariant instruction out of the body and into the header (saving its result in a register for use within the body), then we will eliminate $n - 1$ calculations from the program, a potentially significant savings.

In order to tell whether an instruction is invariant, we need to identify the bodies of loops, and we need to track the locations at which operand values are defined. The first task—identifying loops—is easy in a language that relies exclusively on structured control flow: we simply save appropriate markers when linearizing the syntax tree. In a language with `goto` statements we may need to construct (recover) the loops from a less structured control flow graph.

EXAMPLE 17.21
Data flow equations for reaching definitions

Tracking the locations at which an operand may have been defined amounts to the problem of *reaching definitions*. Formally, we say an instruction that assigns a value v into a location (variable or register) l *reaches* a point p in the code if v may still be in l at p. Like the conversion to static single assignment form, considered informally in Section C-17.4.1, the problem of reaching definitions can be structured as a set of forward, any-path data flow equations. We let Gen_B be the set of final assignments in block B (those that are not overwritten later in B). For each assignment in B we also place in $Kill_B$ all *other* assignments (in any block) to the same location. Then we have

$$\begin{aligned} Out_B &= Gen_B \cup (In_B \setminus Kill_B) \\ In_B &= \bigcup_{\text{predecessors } C \text{ of } B} Out_C \end{aligned}$$

Our initial condition is that $In_1 = \emptyset$: no definitions in the function reach its entry point. Given In_B (the set of reaching definitions at the beginning of the block), we can determine the reaching definitions of all values used *within* B by a simple linear perusal of the code. Because our union operator will iteratively grow the sets of reaching definitions, we begin our computation with $In_B = \emptyset$ for all blocks B (not just B_1). ■

DESIGN & IMPLEMENTATION

17.5 Loop invariants

Many loop invariants arise from address calculations, especially for arrays. Like the common subexpressions discussed in Sidebar C-17.3, they are often not explicit in the program source, and thus cannot be hoisted out of loops by handwritten optimization.

[4] Note that this use of the term is unrelated to the notion of loop invariants in axiomatic semantics (discussed under "Assertions" in Section 4.4).

Given reaching definitions, we define an instruction to be a loop invariant if each of its operands (1) is a constant, (2) has reaching definitions that all lie outside the loop, or (3) has a single reaching definition, even if that definition is an instruction *d* located inside the loop, so long as *d* is itself a loop invariant. (If there is more than one reaching definition for a particular variable, then we cannot be sure of invariance unless we know that all definitions will assign the same value, something that most compilers do not attempt to infer.) As in previous analyses, we begin with the obvious cases and proceed inductively until we reach a fixed point.

EXAMPLE 17.22

Result of hoisting loop invariants

In our `combinations` example, visual inspection of the code reveals two loop invariants: the assignment to `v16` in Block 2 and the assignment to `v42` in Block 3. Moving these invariants out of the loop (and dropping the dead stores and stack pointer updates of Figure C-17.7) yields the code of Figure C-17.10. ■

In the new version of the code, `v16` and `v42` will be calculated even if the loop is executed zero times. In general this precalculation may not be a good idea. If an invariant calculation is expensive and the loop is not in fact executed, then we may have made the program slower. Worse, if an invariant calculation may produce a run-time error (e.g., divide by zero), we may have made the program incorrect. A safe and efficient general solution is to insert an initial test for zero iterations *before* any invariant calculations; we consider this option in Exercise C-17.4. In the specific case of the `combinations` subroutine, our more naive transformation is both safe and (in the common case) efficient.

17.5.2 Induction Variables

An *induction variable* (or register) is one that takes on a simple progression of values in successive iterations of a loop. We will confine our attention here to arithmetic progressions; more elaborate examples appear in Exercises C-17.11 and C-17.12. Induction variables commonly appear as loop indices, subscript computations, or variables incremented or decremented explicitly within the body of the loop. Induction variables are important for two main reasons:

EXAMPLE 17.23

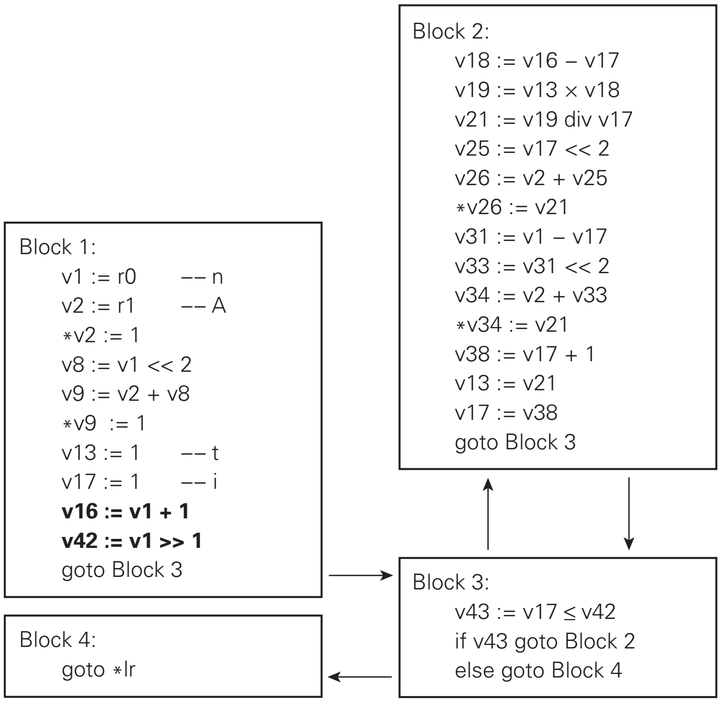
Induction variable strength reduction

- They commonly provide opportunities for strength reduction, most notably by replacing multiplication with addition. For example, if `i` is a loop index variable,

DESIGN & IMPLEMENTATION

17.6 Control flow analysis

Most of the loops in a modern language, with structured control flow, correspond directly to explicit constructs in the syntax tree. A few may be implicit; examples include the loops required to initialize or copy large records or subroutine parameters, or to capture tail recursion. For older languages, the recovery of structure depends on a technique known as *control flow analysis*. A detailed treatment can be found in standard compiler texts [AK02, Sec. 4.5; App97, Sec. 18.1; Muc97, Chap. 7]; we do not discuss it further here.



```
Block 1:
    v1 := r0        -- n
    v2 := r1        -- A
    *v2 := 1
    v8 := v1 << 2
    v9 := v2 + v8
    *v9  := 1
    v13 := 1        -- t
    v17 := 1        -- i
    v16 := v1 + 1
    v42 := v1 >> 1
    goto Block 3
```

```
Block 2:
    v18 := v16 − v17
    v19 := v13 × v18
    v21 := v19 div v17
    v25 := v17 << 2
    v26 := v2 + v25
    *v26 := v21
    v31 := v1 − v17
    v33 := v31 << 2
    v34 := v2 + v33
    *v34 := v21
    v38 := v17 + 1
    v13 := v21
    v17 := v38
    goto Block 3
```

```
Block 3:
    v43 := v17 ≤ v42
    if v43 goto Block 2
    else goto Block 4
```

```
Block 4:
    goto *lr
```

Figure 17.10 **Control flow graph for the `combinations` subroutine after moving the invariant calculations of v16 and v42 (shown in boldface type) out of the loop.** We have also dropped the dead stores of Figure C-17.7, and have eliminated the stack space for t and i, which now reside entirely in registers.

then expressions of the form $t := k \times i + c$ for $i > a$ can be replaced by $t_i := t_{i-1} + k$, where $t_a = k \times a + c$.

EXAMPLE 17.24
Induction variable elimination

- They are commonly redundant: instead of keeping several induction variables in registers across all iterations of the loop, we can often keep a smaller number and calculate the remainder from those when needed (assuming the calculations are sufficiently inexpensive). The result is often a reduction in register pressure with no increase—and sometimes a decrease—in computation cost. In particular, after strength-reducing other induction variables, we can often eliminate the loop index variable itself, with an appropriate change to the end test (see Figure C-17.11 for an example).

The algorithms required to identify, strength-reduce, and possibly eliminate induction variables are more or less straightforward, but fairly tedious [AK02, Sec. 4.5; App97, Sec. 18.3; Muc97, Chap. 14]; we do not present the details here. Similar algorithms can be used to eliminate array and subrange bounds checks in many applications.

```
A : array [1..n] of record
        key : integer
        // other stuff
for i in 1..n
    A[i].key := 0
```

(a)

```
    v1 := 1
    v2 := n
    v3 := sizeof(record)
    v4 := &A − v3
L:  v5 := v1 × v3
    v6 := v4 + v5
    *v6 := 0
    v1 := v1 + 1
    v7 := v1 ≤ v2
    if v7 goto L
```

(b)

```
    v1 := 1
    v2 := n
    v3 := sizeof(record)
    v5 := &A
L:  *v5 := 0
    v5 := v5 + v3
    v1 := v1 + 1
    v7 := v1 ≤ v2
    if v7 goto L
```

(c)

```
    v2 := &A + (n−1) × sizeof(record)
        –– may take >1 instructions
    v3 := sizeof(record)
    v5 := &A
L:  *v5 := 0
    v5 := v5 + v3
    v7 := v5 ≤ v2
    if v7 goto L
```

(d)

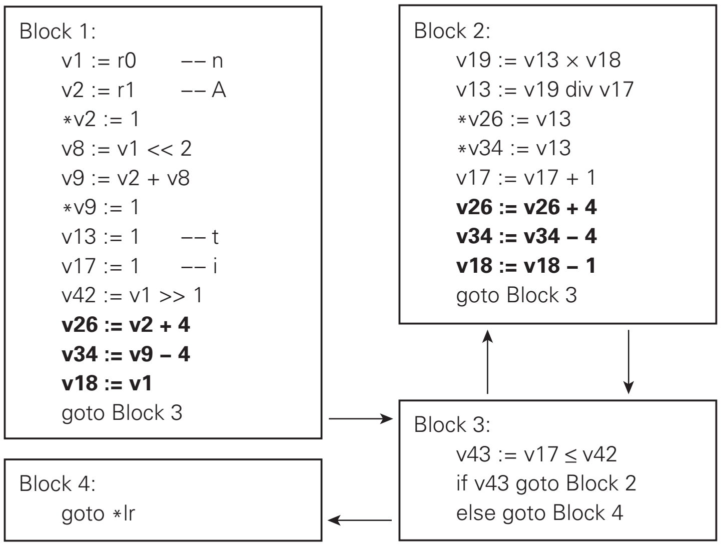
Figure 17.11 **Code improvement of induction variables.** High-level pseudocode source is shown in (a). Target code prior to induction variable optimizations is shown in (b). In (c) we have performed strength reduction on v5, the array index, and eliminated v4, at which point v5 no longer depends on v1 (i). In (d) we have modified the end test to use v5 instead of v1, and have eliminated v1.

EXAMPLE 17.25

Result of induction variable optimization

For our `combinations` example, the code resulting from induction variable optimizations appears in Figure C-17.12. Two induction variables—the array pointers v26 and v34—have undergone strength reduction, eliminating the need for v25, v31, and v33. Similarly v18 has been made independent of v17, eliminating the need for v16. A fifth induction variable—v38—has been eliminated by replacing its single use (the right-hand side of a register–register move) with the addition that computed it. We assume that a repeat of local redundancy elimination in Block 1 has allowed the initialization of v34 to capitalize on the value known to reside in v9.

For presentation purposes, we have also calculated the division operation directly into v13, allowing us to eliminate v21 and its later assignment into v13. A real compiler would probably not make this change until the register allocation phase of compilation, when it would verify that the previous value in v13 is dead at the time of the division (v21 is not an induction variable; its progression of values is not sufficiently simple). Making the change now eliminates the last redun-



```
Block 1:
    v1 := r0       -- n
    v2 := r1       -- A
    *v2 := 1
    v8 := v1 << 2
    v9 := v2 + v8
    *v9 := 1
    v13 := 1       -- t
    v17 := 1       -- i
    v42 := v1 >> 1
    v26 := v2 + 4
    v34 := v9 – 4
    v18 := v1
    goto Block 3
```

```
Block 2:
    v19 := v13 × v18
    v13 := v19 div v17
    *v26 := v13
    *v34 := v13
    v17 := v17 + 1
    v26 := v26 + 4
    v34 := v34 – 4
    v18 := v18 – 1
    goto Block 3
```

```
Block 3:
    v43 := v17 ≤ v42
    if v43 goto Block 2
    else goto Block 4
```

```
Block 4:
    goto *lr
```

Figure 17.12 **Control flow graph for the `combinations` subroutine after optimizing induction variables.** Registers v26 and v34 have undergone strength reduction, allowing v25, v31, and v33 to be eliminated. Registers v38 and v21 have been merged into v17 and v13. The update to v18 has also been simplified, allowing v16 to be eliminated.

dant instruction in the block, and allows us to discuss instruction scheduling in comparative isolation from other issues. ■

17.6 Instruction Scheduling

In the example compiler structure of Figure C-17.1, the next phase after loop optimization is target code generation. As noted in Chapter 15, this phase linearizes the control flow graph and replaces the instructions of the medium-level intermediate form with target machine instructions. The replacements are often driven by an automatically generated pattern-matching algorithm. We will continue to employ our pseudo-assembly "instruction set," so linearization will be the only change we see. Specifically, we will assume that the blocks of the program are concatenated in the order suggested by their names. Control will "fall through" from Block 2 to Block 3, and from Block 3 to Block 4 in the last iteration of the loop.

We will perform two rounds of instruction scheduling separated by register allocation. Given our use of pseudo-assembly, we won't consider peephole optimization in any further detail. In Section C-17.7, however, we will consider additional forms of code improvement for loops that could be applied *prior* to target code generation. We delay discussion of these because the need for them will be clearer after considering instruction scheduling.

On a pipelined machine—particularly one that always executes instructions in program order—performance depends critically on the extent to which the compiler is able to keep the pipeline full. As explained in Section C-5.5.1, delays may result when an instruction (1) needs a functional unit still in use by an earlier instruction, (2) needs data still being computed by an earlier instruction, or (3) cannot even be selected for execution until the outcome or target of a branch has been determined. In this section we consider cases (1) and (2), which can be addressed by reordering instructions within a basic block. A good solution to (3) requires branch prediction, generally with hardware assist. A compiler can solve the subproblem of filling branch delays in a more or less straightforward fashion [Muc97, Sec. 17.1.1].

EXAMPLE 17.26
Remaining pipeline delays

If we examine the body of the loop in our `combinations` example, we find that the optimizations described thus far have transformed Block 2 from the 30 instruction sequence of Figure C-17.3 into the eight-instruction sequence of Figure C-17.12 (not counting the final `goto`s). Unfortunately, on a pipelined machine without instruction reordering, this code is still distinctly suboptimal. In particular, the results of the second and third instructions are used immediately, but the results of multiplies and divides are commonly not available for several cycles. If we assume four-cycle delays, then our block will take 16 cycles to execute.

Dependence Analysis

To schedule instructions to make better use of the pipeline, we first arrange them into a directed acyclic graph (DAG), in which each node represents an instruction, and each arc represents a *dependence*,[5] as described in Section C-5.5.1. Most arcs will represent *flow* dependences, in which one instruction uses a value produced by a previous instruction. A few will represent *anti*-dependences, in which a later instruction overwrites a value read by a previous instruction. In our example, these will correspond to updates of induction variables. If we were performing instruction scheduling after architectural register allocation, then uses of the same register for independent values could increase the number of anti-dependences, and could also induce so-called *output* dependences, in which a later instruction overwrites a value written by a previous instruction. Anti- and output dependences can be hidden on many machines by hardware register renaming (Section C-5.4.3).

EXAMPLE 17.27
Value dependence DAG

Because common subexpression analysis has eliminated all of the loads and stores of `i`, `n`, and `t` in the `combinations` subroutine, and because there are no loads of elements of `A` (only stores), dependence analysis in our example will be dealing solely with values in registers. In general we should need to deal with values in memory as well, and to rely on alias analysis to determine when two instructions might access the same location, and therefore share a dependence. On a target

[5] What we are discussing here is a *dependence DAG*. It is related to, but distinct from, the expression DAG mentioned in Section C-17.3. In particular, the dependence DAG is constructed *after* the assignment of virtual registers to expressions, and its nodes represent instructions, rather than variables and operators.

```
Block 2:                            Scheduled:
 1.  v19 := v13 × v18               v19 := v13 × v18
     —                              v18 := v18 − 1
     —                              —
     —                              —
     —                              —
 2.  v13 := v19 div v17             v13 := v19 div v17
     —                              v17 := v17 + 1
     —                              —
     —                              —
     —                              —
 3.  *v26 := v13                    *v26 := v13
 4.  *v34 := v13                    *v34 := v13
 5.  v17 := v17 + 1                 v26 := v26 + 4
 6.  v26 := v26 + 4                 v34 := v34 − 4
 7.  v34 := v34 − 4
 8.  v18 := v18 − 1
     −− fall through to Block 3

Block 3:
                                      (same)
    v43 := v17 ≤ v42
    if v43 goto Block 2
    −− else fall through to Block 4
```

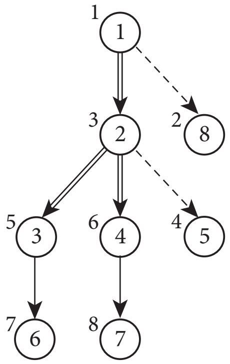


Figure 17.13 Dependence DAG for Block 2 of Figure C-17.12, together with pseudocode for the entire loop, both before (left) and after (right) instruction scheduling. Circled numbers in the DAG correspond to instructions in the original version of the loop. Smaller adjacent numbers give the schedule order in the new loop. Solid arcs indicate flow dependences; dashed arcs indicate anti-dependences. Double arcs indicate pairs of instructions that must be separated by four additional instructions in order to avoid pipeline delays on our hypothetical machine. Delays are shown explicitly in Block 2. Unless we modify the array indexing code (Exercise C-17.20), only two instructions can be moved.

machine with condition codes (i.e., most machines today—see Section C-5.3), we should need to model these explicitly, tracking flow, anti-, and output dependences.

The dependence DAG for Block 2 of our `combinations` example appears in Figure C-17.13. In this case the DAG turns out to be a tree. It was generated by examining the code from top to bottom, linking each instruction *i* to each subsequent instruction *j* such that *j* reads a register written by *i* (solid arcs) or writes a register read by *i* (dashed arcs).

Any topological sort of a dependence DAG (i.e., any enumeration of the nodes in which each node appears before its children) will represent a correct schedule. Ideally we should like to choose a sort that minimizes overall delay. As with many aspects of code improvement, this task is NP-hard, so practical techniques rely upon heuristics.

To capture timing information, we define a function *latency*(i, j) that returns the number of cycles that must elapse between the scheduling of instructions *i* and *j* if *j* is to run after *i* in the same pipeline without stalling. (To maintain machine independence, this portion of the code improver must be driven by tables of machine characteristics; those characteristics must not be "hard-coded.") Nontrivial latencies can result from data dependences or from conflicts for use of some physical resource, such as an incompletely pipelined functional unit. We will assume in our example that all units are fully pipelined, so all latencies are due to data dependences.

We now traverse the DAG from the roots down to the leaves. At each step we first determine the set of *candidate* nodes: those for which all parents have been scheduled. For each candidate *i* we then use the *latency* function with respect to already-scheduled nodes to determine the earliest time at which *i* could execute without stalling. We also precalculate the maximum over all paths from *i* to a leaf of the sums of the latencies on arcs; this gives us a lower bound on the time that will be required to finish the basic block after *i* has been scheduled. In our examples we will use the following three heuristics to choose among candidate nodes:

1. Favor nodes that can be started without stalling.
2. If there is a tie, favor nodes with the maximum delay to the end of the block.
3. If there is still a tie, favor the node that came first in the original source code (this strategy leads to more intuitive assembly language, which can be helpful in debugging).

Other possible scheduling heuristics include:

- Favor nodes that have a large number of children in the DAG (this increases flexibility for future iterations of the scheduling algorithm).
- Favor nodes that are the final use of a register (this reduces register pressure).
- If there are multiple pipelines, favor nodes that can use a pipeline that has not received an instruction recently.

If our target machine has multiple pipelines, then we must keep track for each instruction of the pipeline we think it will use, so we can distinguish between candidates that can start in the current cycle and those that cannot start until the next. (Imprecise machine models, cache misses, or other unpredictable delays may cause our guess to be wrong some of the time.)

EXAMPLE 17.28

Result of instruction scheduling

Unfortunately, our example DAG leaves very little room for choice. The only possible improvements are to move Instruction 8 into one of the multiply or divide delay slots and Instruction 5 into one of the divide delay slots, reducing the total cycle count of Block 2 from 16 to 14. If we assume (1) that our target machine correctly predicts a backward branch at the bottom of the loop, and (2) that we can replicate the first instruction of Block 2 into a nullifying delay slot of the branch, then we incur no additional delays in Block 3 (except in the last iteration). The overall duration of the loop is therefore 18 cycles per iteration before scheduling, 16 cycles per iteration after scheduling—an improvement of 11%. In Section C-17.7 we will consider other versions of the block, in which rescheduling yields significantly faster code. ■

As noted near the end of Section C-17.1, we shall probably want to repeat instruction scheduling after global code improvement and register allocation. If there are times when the number of virtual registers with useful values exceeds the number of architectural registers on the target machine, then we shall need to generate code to *spill* some values to memory and load them back in again later. Rescheduling will be needed to handle any delays induced by the loads.

✓ **CHECK YOUR UNDERSTANDING**

19. What is a *loop invariant*? A *reaching definition*?
20. Why might it sometimes be unsafe to hoist an invariant out of a loop?
21. What are *induction variables*? What is *strength reduction*?
22. What is *control flow analysis*? Why is it less important than it used to be?
23. What is *register pressure*? *Register spilling*?
24. Is instruction scheduling a machine-independent code improvement technique? Explain.
25. Describe the creation and use of a *dependence DAG*. Explain the distinctions among *flow*, *anti-*, and *output* dependences.
26. Explain the tension between instruction scheduling and register allocation.
27. List several heuristics that might be used to prioritize instructions to be scheduled.

17.7 Loop Improvement II

As noted in Section C-17.5, code improvements that improve the speed of loops are particularly important, because loops are where most programs spend most of their time. In this section we consider transformations that improve instruction scheduling by restructuring a loop body to include portions of more than one iteration of the original loop, and that manipulate multiply nested loops to improve cache performance or increase opportunities for parallelization. Extensive coverage of loop transformations and dependence theory can be found in Allen and Kennedy's text [AK02].

17.7.1 Loop Unrolling and Software Pipelining

EXAMPLE 17.29
Result of loop unrolling

Loop *unrolling* is a transformation that embeds two or more iterations of a source-level loop in a single iteration of a new, longer loop, allowing the scheduler to intermingle the instructions of the original iterations. If we unroll two iterations of our combinations example we obtain the code of Figure C-17.14. We have used separate names (here starting with the letter 't') for registers written in the initial half of the loop. This convention minimizes anti- and output dependences, giving us more latitude in scheduling. In an attempt to minimize loop overhead, we have also recognized that the array pointer induction variables (v26 and v34) need only be updated once in each iteration of the loop, provided that we use displacement addressing in the second set of store instructions. The new instructions added to the end of Block 1 cover the case in which *n* div 2, the number of iterations of the original loop, is not an even number.

Again assuming that the branch in Block 3 can be scheduled without delays, the total time for our unrolled loop (prior to scheduling) is 32 cycles, or 16 cycles per iteration of the original loop. After scheduling, this number is reduced to 12 cycles per iteration of the original loop. Unfortunately, eight cycles (four per original iteration) are still being lost to stalls. ■

EXAMPLE 17.30

Result of software pipelining

If we unroll the loop three times instead of two (see Exercise C-17.21), we can bring the cost (with rescheduling) down to 11.3 cycles per original iteration, but this is not much of an improvement. The basic problem is illustrated in the top half of Figure C-17.15. In the original version of the loop, the two store instructions cannot begin until after the divide delay. If we unroll the loop, then instructions of the internal iterations can be intermingled, but six cycles of "shut-down" cost (four delay slots and two stores) are still needed after the final divide.

A *software-pipelined* version of our combinations subroutine appears schematically in the bottom half of Figure C-17.15, and as a control flow graph in Figure C-17.16. The idea is to build a loop whose body comprises portions of several consecutive iterations of the original loop, with no internal start-up or shut-down cost. In our example, each iteration of the software-pipelined loop contributes to three separate iterations of the original loop. Within each new iteration (shown between vertical bars) nothing needs to wait for the divide to complete. To avoid delays, we have altered the code in several ways. First, because each iteration of the new loop contributes to several iterations of the original loop, we must ensure that there are enough iterations to run the new loop at least once (this is the purpose of the test in the new Block 1). Second, we have preceded and followed the loop with code to "prime" and "flush" the "pipeline": to execute the early portions of the first iteration and the final portions of the last few. As we did when unrolling the loop, we use a separate name (t13 in this case) for any register written in the new "pipeline flushing" code. Third, to minimize the amount of priming required we have initialized v26 and v34 one slot before their original positions, so that the first iteration of the pipelined loop can "update" them as part of a "zero-th" original iteration. Finally, we have dropped the initialization of v13 in Block 1: our priming code has left that register dead at the end of the block. (Live variable analysis on virtual registers could have been used to discover this fact.)

Both the original and pipelined versions of the loop carry five nonconstant values across the boundary between iterations, but one of these has changed identity: whereas the original loop carried the result of the divide around to the next multiply

```
Block 1:
        ...                   -– code from Block 1, figure 16.11
        v44 := v42 & 01
        if !v44 goto Block 3
        -– else fall through to Block 1a
Block 1a:
        *v26 := 1
        *v34 := 1
        v17 := 2
        v26 := v26 + 4
        v34 := v34 − 4
        v18 := v18 − 1
        goto Block 3
Block 2:                         Scheduled:
   1.   t19 := v13 × v18            t19 := v13 × v18
        —                           t18 := v18 − 1
        —                           t17 := v17 + 1
        —                           v18 := t18 − 1
        —                           —
   2.   t13 := t19 div v17          t13 := t19 div v17
        —                           v17 := t17 + 1
        —                           —
        —                           —
        —                           —
   3.   *v26 := t13                 v19 := t13 × t18
   4.   *v34 := t13                 *v26 := t13
   5.   t17 := v17 + 1              *v34 := t13
   6.   v26 := v26 + 8              v26 := v26 + 8
   7.   v34 := v34 − 8              v34 := v34 − 8
   8.   t18 := v18 − 1              v13 := v19 div t17
   9.   v19 := t13 × t18            —
        —                           —
        —                           —
        —                           —
        —                           *(v26−4) := v13
  10.   v13 := v19 div t17          *(v34+4) := v13
        —
        —
        —
        —
  11.   *(v26−4) := v13
  12.   *(v34+4) := v13
  13.   v17 := t17 + 1
  14.   v18 := t18 − 1
        -– fall through to Block 3
Block 3:
                                     (same)
        v43 := v17 ≤ v42
        if v43 goto Block 2
        -– else fall through to Block 4
```

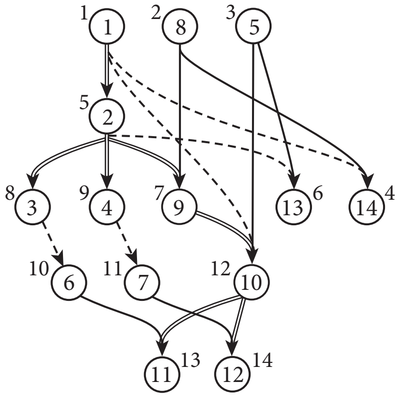


Figure 17.14 Dependence DAG for Block 2 of the `combinations` subroutine after unrolling two iterations of the body of the loop. Also shown is linearized pseudocode for the entire loop, both before (left) and after (right) instruction scheduling. New instructions added to the end of Block 1 cover the case in which the number of iterations of the original loop is not a multiple of two.

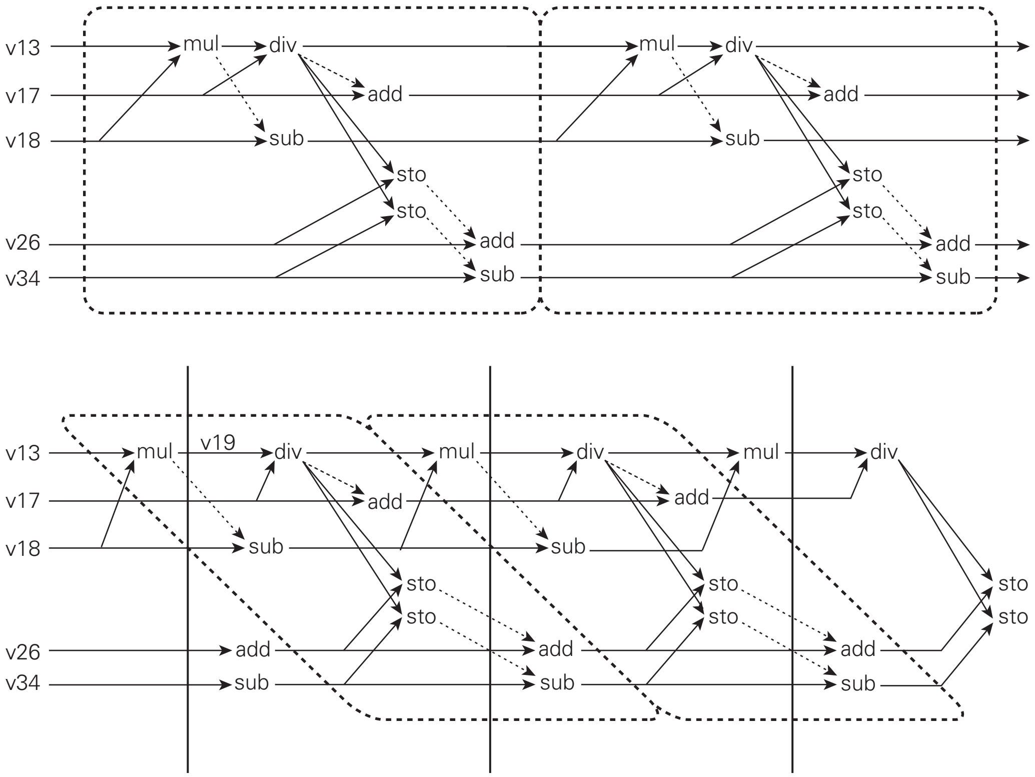
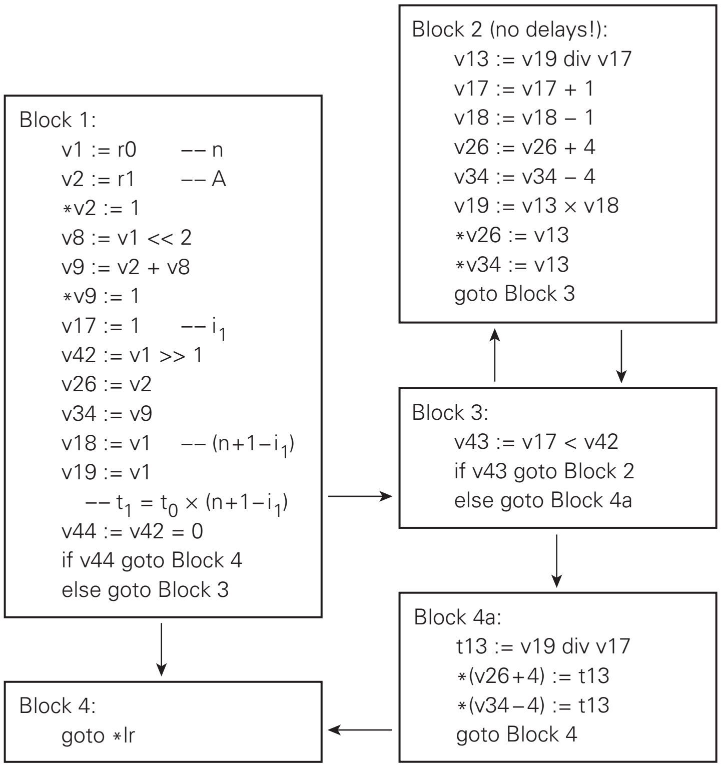


Figure 17.15 **Software pipelining.** The top diagram illustrates the execution of the original (nonpipelined) loop. In the bottom diagram, each iteration of the original loop has been spread across three iterations of the pipelined loop. Iterations of the original loop are enclosed in a dashed-line box; iterations of the pipelined loop are separated by solid vertical lines. In the bottom diagram we have also shown the code to prime the pipeline prior to the first iteration, and to flush it after the last.

in register `v13`, the pipelined loop carries the result of the multiply forward to the divide in register `v19`. In more complicated loops it may be necessary to carry two or even three versions of a single register (corresponding to two or more iterations of the original loop) across the boundary between iterations of the pipelined loop. We must invent new virtual registers (similar to the new `t13` and to the `t` registers in the unrolled version of the `combinations` example) to hold the extra values. In such a case software pipelining has the side effect of increasing register pressure.

Each of the instructions in the loop of the pipelined version of the `combinations` subroutine can proceed without delay. The total number of cycles per iteration has been reduced to 10. We can do even better if we combine loop unrolling and software pipelining. For example, by embedding two multiply–divide pairs in each iteration (drawn, with their accompanying instructions, from four iterations of the original loop, rather than just three), we can update the array



```
Block 1:
      v1 := r0       -- n
      v2 := r1       -- A
      *v2 := 1
      v8 := v1 << 2
      v9 := v2 + v8
      *v9 := 1
      v17 := 1       -- i1
      v42 := v1 >> 1
      v26 := v2
      v34 := v9
      v18 := v1      -- (n+1−i1)
      v19 := v1
         -- t1 = t0 × (n+1−i1)
      v44 := v42 = 0
      if v44 goto Block 4
      else goto Block 3
```

```
Block 2 (no delays!):
      v13 := v19 div v17
      v17 := v17 + 1
      v18 := v18 − 1
      v26 := v26 + 4
      v34 := v34 − 4
      v19 := v13 × v18
      *v26 := v13
      *v34 := v13
      goto Block 3
```

```
Block 3:
      v43 := v17 < v42
      if v43 goto Block 2
      else goto Block 4a
```

```
Block 4a:
      t13 := v19 div v17
      *(v26+4) := t13
      *(v34−4) := t13
      goto Block 4
```

```
Block 4:
      goto *lr
```

Figure 17.16 Control flow graph for the `combinations` subroutine after software pipelining. The additional code and test at the end of Block 1, the change to the test in Block 3 (< instead of ≤), and the new block (4a) make sure that there are enough iterations to accommodate the pipeline, prime it with the beginnings of the initial iteration, and flush the end of the final iteration. Suffixes on variable names in the comments in Block 1 refer to loop iterations: t_1 is the value of t in the first iteration of the loop; t_0 is a "zero-th" value used to prime the pipeline.

pointers and check the termination condition half as often, for a net of only eight cycles per iteration of the original loop (see Exercise C-17.22).

To summarize, loop unrolling serves to reduce loop overhead, and can also increase opportunities for instruction scheduling. Software pipelining does a better job of facilitating scheduling, but does not address loop overhead. A reasonable code improvement strategy is to unroll loops until the per-iteration overhead falls below some acceptable threshold of the total work, then employ software pipelining if necessary to eliminate scheduling delays.

17.7.2 Loop Reordering

The code improvement techniques that we have considered thus far have served two principal purposes: to eliminate redundant or unnecessary instructions, and to minimize stalls on a pipelined machine. Two other goals have become increasingly important over the years. First, as improvements in processor speed have outstripped improvements in memory latency, it has become increasingly important to minimize cache misses. Second, for parallel machines, it has become important to identify sections of code that can execute concurrently. As with other optimizations, the largest benefits come from changing the behavior of loops. We touch on some of the issues here; suggestions for further reading can be found at the end of the chapter.

Cache Optimizations

EXAMPLE 17.31
Loop interchange

Probably the simplest example of cache optimization can be seen in code that traverses a multidimensional matrix (array):

```
for i := 1 to n
    for j := 1 to n
        A[i, j] := 0
```

If A is laid out in row-major order, and if each cache line contains m elements of A, then this code will suffer n^2/m cache misses. On the other hand, if A is laid out in column-major order, and if the cache is too small to hold n lines of A, then the code will suffer n^2 misses, fetching the entire array from memory m times. The difference can have an enormous impact on performance. A loop-reordering compiler can improve this code by *interchanging* the nested loops:

```
for j := 1 to n
    for i := 1 to n
        A[i, j] := 0
```

■

EXAMPLE 17.32
Loop tiling (blocking)

In more complicated examples, interchanging loops may improve locality of reference in one array but worsen it in others. Consider this code to transpose a two-dimensional matrix:

```
for j := 1 to n
    for i := 1 to n
        A[i, j] := B[j, i]
```

If A and B are laid out the same way in memory, one of them will be accessed along cache lines, but the other will be accessed across them. In this case we may improve locality of reference by *tiling* or *blocking* the loops:

```
for it := 1 to n by b
    for jt := 1 to n by b
        for i := it to min(it + b − 1, n)
            for j := jt to min(jt + b − 1, n)
                A[i, j] := B[j, i]
```

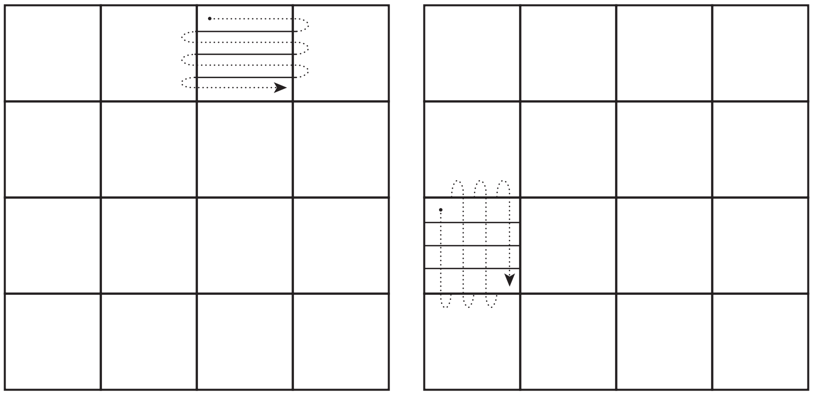


Figure 17.17 **Tiling (blocking) of a matrix operation.** As long as one tile of A and one tile of B can fit in the cache simultaneously, only one access in m will cause a cache miss (where m is the number of elements per cache line).

Here the min calculations cover the possibility that b does not divide n evenly. They can be dropped if n is known to be a multiple of b. Alternatively, if we are willing to replicate the code inside the innermost loop, then we can generate different code for the final iteration of each loop (Exercise C-17.25).

The new code iterates over $b \times b$ blocks of A and B, one in row-major order, the other in column-major order, as shown in Figure C-17.17. If we choose b to be a multiple of m such that the cache can hold two $b \times b$ blocks of data simultaneously, then both A and B will suffer only one cache miss per m array elements, fetching everything from memory exactly once.[6] Tiling is useful in a wide variety of algorithms on multidimensional arrays. Exercise C-17.23 considers matrix multiplication. ■

Two other transformations that may sometimes improve cache locality are loop *distribution* (also called *fission* or *splitting*), and its inverse, loop *fusion* (also known as *jamming*). Distribution splits a single loop into multiple loops, each of which contains some fraction of the statements of the original loop. Fusion takes separate loops and combines them.

EXAMPLE 17.33
Loop distribution

Consider, for example, the following code to reorganize a pair of arrays:

```
for i := 0 to n−1
    A[i] := B[M[i]];
    C[i] := D[M[i]];
```

[6] Although A is being written, not read, the hardware will fetch each line of A from memory on the first write to the line, so that the single modified element can be updated within the cache. The hardware has no way to know that the entire line will be modified before it is written back to memory.

Here M defines a mapping from locations in B or D to locations in A or C. If either B or D, but not both, can fit into the cache at once, then we may get faster code through distribution:

```
for i := 1 to n
    A[i] := B[M[i]];
for i := 1 to n
    C[i] := D[M[i]];
```

■

EXAMPLE 17.34

Loop fusion

On the other hand, in the following code, separate loops may lead to *poorer* locality:

```
for i := 1 to n
    A[i] := A[i] + c
for i := 1 to n
    if A[i] < 0 then A[i] := 0
```

If A is too large to fit in the cache in its entirety, then these loops will fetch the entire array from memory twice. If we fuse them, however, we need only fetch A once:

```
for i := 1 to n
    A[i] := A[i] + c
    if A[i] < 0 then A[i] := 0
```

■

If two loops do not have identical bounds, it may still be possible to fuse them if we transform induction variables or *peel* some constant number of iterations off of one of the loops.

EXAMPLE 17.35

Obtaining a perfect loop nest

Loop distribution may serve to facilitate other transformations (e.g., loop interchange) by transforming an "imperfect" loop nest into a "perfect" one:

```
for i := 1 to n
    A[i] := A[i] + c
    for j := 1 to n
        B[i, j] := B[i, j] × A[i]
```

This nest is called imperfect because the outer loop contains more than just the inner loop. Distribution yields two outermost loops:

```
for i := 1 to n
    A[i] := A[i] + c
for i := 1 to n
    for j := 1 to n
        B[i, j] := B[i, j] × A[i]
```

The nested loops are now perfect, and can be interchanged if desired. ■

In keeping with our earlier discussions of loop optimizations, we note that loop distribution can reduce register pressure, while loop fusion can reduce loop overhead.

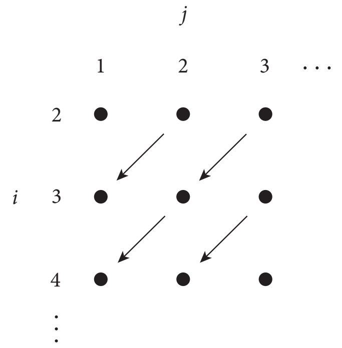
Loop Dependences

EXAMPLE 17.36
Loop-carried dependences

When reordering loops, we must be extremely careful to respect all data dependences. Of particular concern are so-called *loop-carried* dependences, which constrain the orders in which iterations can occur. Consider, for example, the following:

```
for i := 2 to n
    for j := 1 to n−1
        A[i, j] := A[i, j] − A[i−1, j+1]
```

Here the calculation of A[i, j] in iteration (i, j) depends on the value of A[i−1, j+1], which was calculated in iteration $(i-1, j+1)$. This dependence is often represented by a diagram of the *iteration space*:



The *i* and *j* dimensions in this diagram represent loop indices, *not* array subscripts. The arcs represent the loop-carried flow dependence.

If we wish to interchange the *i* and *j* loops of this code (e.g., to improve cache locality), we find that we cannot do it, because of the dependence: we would end up trying to write A[i, j] before we had written A[i−1, j+1]. ■

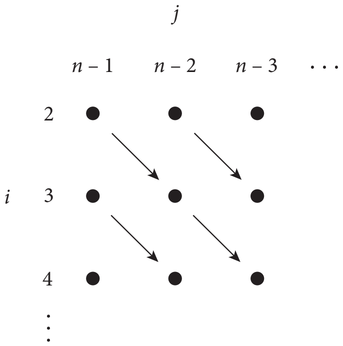
To analyze loop-carried dependences, high-performance optimizing compilers use symbolic mathematics to characterize the sets of index values that may cause the subscript expressions in different array references to evaluate to the same value. Compilers differ somewhat in the sophistication of this analysis. Most can handle linear combinations of loop indices. None, of course, can handle all expressions, since equivalence of general formulae is undecidable. When unable to fully characterize subscripts, a compiler must conservatively assume the worst, and rule out transformations whose safety cannot be proven.

EXAMPLE 17.37
Loop reversal and interchange

In many cases a loop with a fully characterized dependence that precludes a desired transformation can be modified in a way that eliminates the dependence. In Example C-17.36 above, we can *reverse* the order of the *j* loop without violating the dependence:

```
for i := 2 to n
    for j := n−1 to 1 by−1
        A[i, j] := A[i, j] − A[i−1, j+1]
```

This change transforms the iteration space:



And now the loops can safely be interchanged:

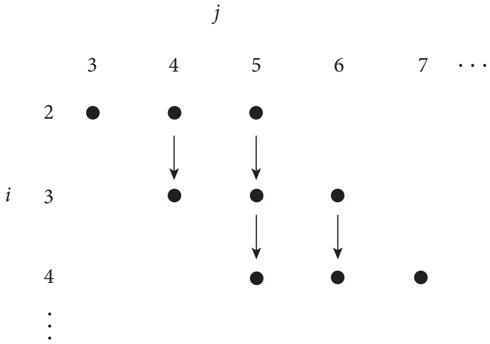
```
for j := n−1 to 1 by−1
    for i := 2 to n
        A[i, j] := A[i, j] − A[i−1, j+1]
```

EXAMPLE 17.38
Loop skewing

Another transformation that sometimes serves to eliminate a dependence is known as loop *skewing*. In essence, it reshapes a rectangular iteration space into a parallelogram, by adding the outer loop index to the inner one, and then subtracting from the appropriate subscripts:

```
for i := 2 to n
    for j := i+1 to i+n−1
        A[i, j−i] := A[i, j−i] − A[i−1, j+1−i]
```

A moment's consideration will reveal that this code accesses the exact same elements as before, in the exact same order. Its iteration space, however, looks like this:



Now the loops can safely be interchanged. The transformation is complicated by the need to accommodate the sloping sides of the iteration space. To avoid using min and max functions, we can divide the space into two triangular sections, each of which has its own loop nest:

```
for j := 3 to n+1
    for i := 2 to j−1
        A[i, j− i] := A[i, j− i] − A[i−1, j+1− i]
for j := n+2 to 2×n−1
    for i := j−n+1 to n
        A[i, j− i] := A[i, j− i] − A[i−1, j+1− i]
```

Skewing has led to more complicated code than did reversal of the *j* loop, but it could be used in the presence of other dependences that would eliminate reversal as an option.

Several other loop transformations, including distribution, can also be used in certain cases to eliminate loop-carried dependences, allowing us to apply techniques that improve cache locality or (as discussed immediately below) enable us to execute code in parallel on a vector or multicore machine. Of course, no set of transformations can eliminate all dependences; some code simply can't be improved.

Parallelization

Loop iterations (at least in nonrecursive programs) constitute the principal source of operations that can execute in parallel. Ideally, one needs to find *independent* loop iterations: ones with no loop-carried dependences. (In some cases, iterations can also profitably be executed in parallel even if they have dependences, so long as they synchronize their operations appropriately.) In Example 13.8 and Section 13.4.6 we considered loop constructs that allow the programmer to specify parallel execution. Even in languages without such special constructs, a compiler can often *parallelize* code by identifying—or creating—loops with as few loop-carried dependences as possible. The transformations described above are valuable tools in this endeavor.

EXAMPLE 17.39
Coarse-grain parallelization

Given a parallelizable loop, the compiler must consider several other issues in order to ensure good performance. One of the most important of these is the *granularity* of parallelism. For a very simple example, consider the problem of "zeroing out" a two-dimensional array, here indexed from 0 to $n-1$ in each dimension, and laid out in row-major order:

```
for i := 0 to n−1
    for j := 0 to n−1
        A[i, j] := 0
```

On a machine comprising several general-purpose processor cores, we would probably parallelize the outer loop:

```
–– on processor core pid:
for i := (n/p × pid) to (n/p × (pid + 1) − 1)
    for j := 1 to n
        A[i, j] := 0
```

Here we have given each core a band of rows to initialize. We have assumed that cores are numbered from 0 to $p-1$, and that p divides n evenly.

The strategy on a vector machine is very different. Such a machine includes a collection of v-element vector registers, and instructions to load, store, and compute on vector data. The vector instructions are deeply pipelined, allowing the machine to exploit a high degree of *fine-grain* parallelism. To satisfy the hardware, the compiler needs to parallelize *inner* loops:

EXAMPLE 17.40
Strip mining

```
for i := 0 to n−1
    for j := 0 to n−1 by v
        A[i, j:j+v−1] := 0        −− vector operation
```

Here the notation A[i, j:j+v−1] represents a v-element *slice* of A. The constant v should be set to the length of a vector register (which we again assume divides n evenly). The code transformation that extracts v-element operations from longer loops is known as *strip mining*. It is essentially a one-dimensional form of tiling. ■

Other issues of importance in parallelizing compilers include *communication* and *load balance*. Just as locality of reference reduces communication between the cache and main memory on a single-core machine, locality in parallel programs reduces communication among cores and between the cores and memory. Optimizations similar to those employed to reduce the number of cache misses on a single-core machine can be used to reduce communication traffic on a multicore machine.

Load balance refers to the division of labor among processor cores. If we divide the work of a program among 16 cores, we shall obtain a speedup of close to 16 only if each core takes the same amount of time to do its work. If we accidentally assign 5% of the work to each of 15 cores and 25% of the work to the 16th, we are likely to see a speedup of no more than 4×. For simple loops it is often possible to predict performance accurately enough to divide the work among cores at compile time. For more complex loops, in which different iterations perform different amounts of work or have different cache behavior, it is often better to generate *self-scheduled* code, which divides the work up at run time. In its simplest form, self scheduling creates a “bag of tasks,” as described in Section 13.2. Each task consists of a set of loop iterations. The number of such tasks is chosen to be significantly larger than the number of cores. When finished with a given task, a core goes back to the bag to get another.

17.8 Register Allocation

In a simple compiler with no global optimizations, register allocation can be performed independently in every basic block. To avoid the obvious inefficiency of storing frequently accessed variables to memory at the end of many blocks, and reading them back in again in others, simple compilers usually apply a set of heuristics to identify such variables and allocate them to registers over the life of a subroutine. Obvious candidates for a dedicated register include loop indices and scalar local variables and parameters.

It has been known since the early 1970s that register allocation is equivalent to the NP-hard problem of graph coloring. Following the work of Chaitin et al. [CAC$^+$81], heuristic (nonoptimal) implementations of graph coloring have become a common approach to register allocation in aggressive optimizing compilers. We describe the basic idea here; for more detail see Cooper and Torczon's text [CT11, Chap. 13].

EXAMPLE 17.41
Live ranges of virtual registers

The first step is to identify virtual registers that *cannot* share an architectural register, because they contain values that are live concurrently. To accomplish this step we use reaching definitions data flow analysis (Section C-17.5.1). For the software-pipelined version of our `combinations` subroutine (Figure C-17.16), we can chart the *live ranges* of the virtual registers as shown in Figure C-17.18. Note that the live range of `v19` spans the backward branch at the end of Block 2; though typographically disconnected it is contiguous in time. ■

EXAMPLE 17.42
Register coloring

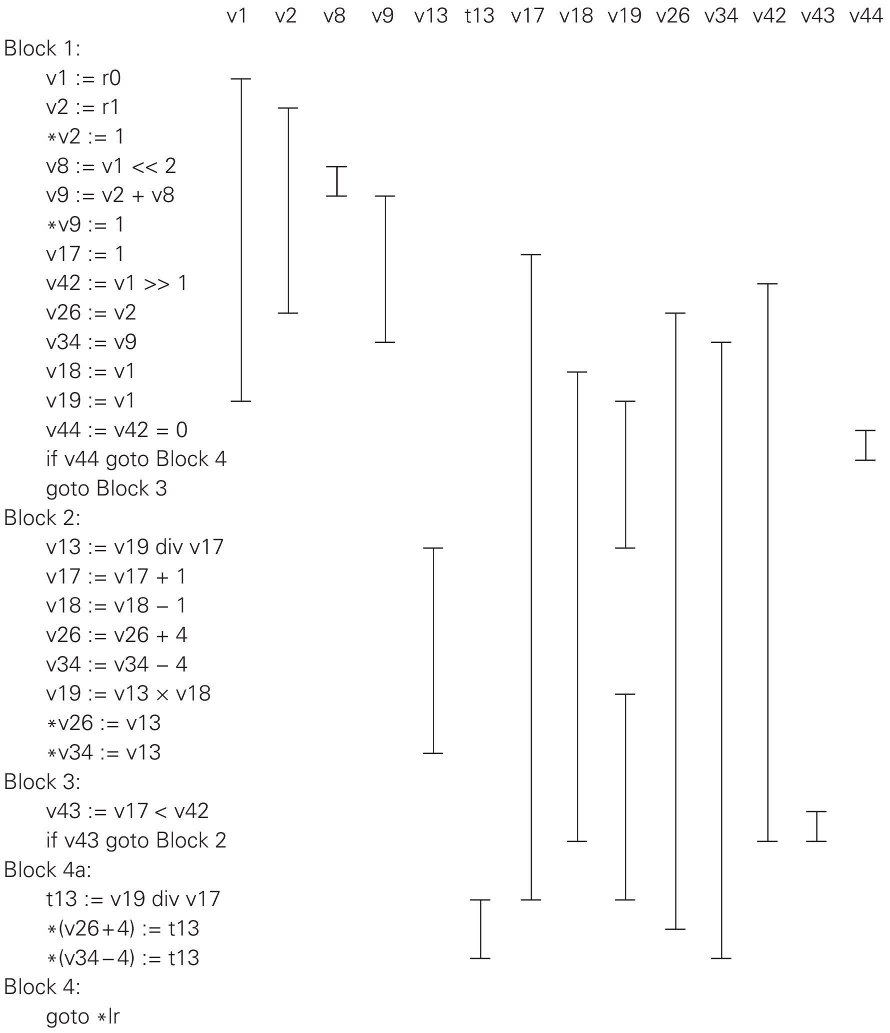
Given these live ranges, we construct a *register interference graph*. The nodes of this graph represent virtual registers. Registers `v`i and `v`j are connected by an arc if they are simultaneously live. The interference graph corresponding to Figure C-17.18 appears in Figure C-17.19. The problem of mapping virtual registers onto the smallest possible number of architectural registers now amounts to finding a *minimal coloring* of this graph: an assignment of "colors" to nodes such that no arc connects two nodes of the same color.

In our example, we can find one of several optimal solutions by inspection. The six registers in the center of the figure constitute a clique (a completely connected subgraph); each must be mapped to a separate architectural register. Moreover there are three cases—registers `v1` and `v19`, `v2` and `v26`, and `v9` and `v34`—in which one register is copied into the other somewhere in the code, but the two are never simultaneously live. If we use a common architectural register in each of these cases then we can eliminate the copy instructions; this optimization is known as *live range coalescing*. Registers `v13`, `v43`, and `v44` are connected to every member of the clique, but not to each other; they can share a seventh architectural register. Register `v8` is connected to `v1`, `v2`, and `v9`, but not to anything else; we have arbitrarily chosen to have both it and `t13` share with the three registers on the right. ■

EXAMPLE 17.43
Optimized `combinations` subroutine

Final code for the `combinations` subroutine appears in Figure C-17.20. We have left `v1`/`v19` and `v2`/`v26` in `r0` and `r1`, the registers in which their initial values were passed. Because our subroutine is a leaf, these registers are never needed for other arguments. Following Arm conventions (Section C-5.4.5), we have used registers `r2` through `r6` as additional temporary registers. Of these, `r4` through `r6` are callee-saves, so we have pushed their old values in the prologue and popped them in the epilogue. ■

We have glossed over two important issues. First, on almost any real machine, architectural registers are not uniform. Integer registers cannot be used for floating-point operations. Caller-saves registers should not be used for variables whose values are needed across subroutine calls. Registers that are overwritten by special instructions (e.g., byte string search on a CISC machine) should not be used to hold values that are needed across such instructions. To handle constraints of this type, the register interference graph is usually extended to contain nodes for



```
Block 1:
    v1 := r0
    v2 := r1
    *v2 := 1
    v8 := v1 << 2
    v9 := v2 + v8
    *v9 := 1
    v17 := 1
    v42 := v1 >> 1
    v26 := v2
    v34 := v9
    v18 := v1
    v19 := v1
    v44 := v42 = 0
    if v44 goto Block 4
    goto Block 3
Block 2:
    v13 := v19 div v17
    v17 := v17 + 1
    v18 := v18 − 1
    v26 := v26 + 4
    v34 := v34 − 4
    v19 := v13 × v18
    *v26 := v13
    *v34 := v13
Block 3:
    v43 := v17 < v42
    if v43 goto Block 2
Block 4a:
    t13 := v19 div v17
    *(v26+4) := t13
    *(v34−4) := t13
Block 4:
    goto *lr
```

Figure 17.18 Live ranges for virtual registers in the software-pipelined version of the `combinations` subroutine (Figure C-17.16).

both virtual and architectural registers. Arcs are then drawn from each virtual register to the architectural registers to which it should not be mapped. Each architectural register is also connected to every other, to force them all to have separate colors. After coloring the resulting graph, we assign each virtual register to the architectural register of the same color. On Arm (for which we are supposedly generating code), v43 and v44 must actually be mapped to the condition codes in the processor status register (psr). The astute reader may have noticed that we did

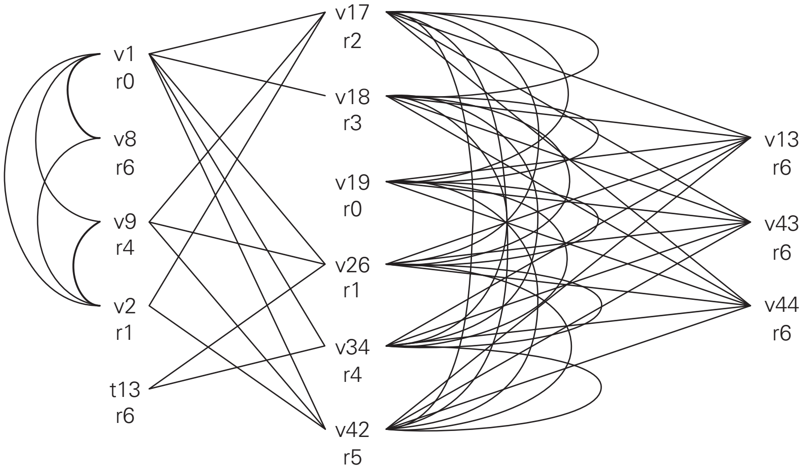


Figure 17.19 Register interference graph for the software pipelined version of the `combinations` subroutine. Using architectural register names, we have indicated one of several possible seven-colorings.

```
Block 1:
    push {r4, r5, r6}
    ∗r1 := 1
    r6 := r0 << 2
    r4 := r1 + r6
    ∗r4 := 1
    r2 := 1
    r5 := r0 >> 1
    r3 := r0
    cc := r5 = 0
    if cc goto Block 4
    goto Block 3
Block 2:
    r6 := r0 div r2
    r2 := r2 + 1
    r3 := r3 − 1
    r1 := r1 + 4
    r4 := r4 − 4
    r0 := r6 × r3
    ∗r1 := r6
    ∗r4 := r6
Block 3:
    cc := r2 < r5
    if cc goto Block 2
Block 4a:
    r6 := r0 div r2
    ∗(r1+4) := r6
    ∗(r4−4) := r6
Block 4:
    pop {r4, r5, r6}
    goto ∗lr
```

Figure 17.20 Final code for the `combinations` subroutine, after assigning architectural registers and eliminating useless copy instructions.

so in Figure C-17.20. In our particular example, the change has no impact on the number of colors required for the remaining virtual registers.

The second issue we've ignored is what happens when there aren't enough architectural registers to go around. In this case it will not be possible to color the interference graph. Using a variety of heuristics (which we do not cover here), the compiler chooses virtual registers whose live ranges can be *split* into two or more subranges. A value that is live at the end of a subrange may be *spilled* (stored) to memory, and reloaded at the beginning of the subsequent subrange. Alternatively, it may be *rematerialized* by repeating the calculation that produced it (assuming the necessary operands are still available). Which is cheaper will depend on the cost of loads and stores and the complexity of the generating calculation.

It is easy to prove that with a sufficient number of range splits it is possible to color any graph, given at least three colors. The trick is to find a set of splits that keeps the cost of spills and rematerialization low. Once register allocation is complete, as noted in Sections C-17.1 and C-17.6, we shall want to repeat instruction scheduling, in order to fill any newly created load delays.

CHECK YOUR UNDERSTANDING

28. What is the difference between *loop unrolling* and *software pipelining*? Explain why the latter may increase register pressure.
29. What is the purpose of *loop interchange*? *Loop tiling* (*blocking*)?
30. What are the potential benefits of *loop distribution*? *Loop fusion*? What is *loop peeling*?
31. What does it mean for loops to be *perfectly nested*? Why are perfect loop nests important?
32. What is a *loop-carried dependence*? Describe three loop transformations that may serve in some cases to eliminate such a dependence.
33. Describe the fundamental difference between the parallelization strategy for multicore machines and the parallelization strategy for vector machines.
34. What is *self scheduling*? When is it desirable?
35. What is the *live range* of a register? Why might it not be a contiguous range of instructions?
36. What is a *register interference graph*? What is its significance? Why do production compilers depend on heuristics (rather than precise solutions) for register allocation?
37. List three reasons why it might not be possible to treat architectural registers uniformly for purposes of register allocation.

17.9 Summary and Concluding Remarks

This chapter has addressed the subject of code improvement (“optimization”). We considered several of the most important optimization techniques, including peephole optimization; local and global (procedure-level) redundancy elimination (constant folding, constant propagation, copy propagation, common subexpression elimination); loop improvement (invariant hoisting, strength reduction or elimination of induction variables, unrolling and software pipelining, reordering for cache improvement or parallelization); instruction scheduling; and register allocation. Many others techniques, too numerous to mention, can be found in the literature or in production use.

To facilitate code improvement, we introduced several new data structures and program representations, including dependence DAGs (for instruction scheduling), static single-assignment (SSA) form (for many purposes, including global common subexpression elimination via value numbering), and the register interference graph (for architectural register allocation). For many global optimizations we made use of data flow analysis. Specifically, we employed it to identify available expressions (for global common subexpression elimination), to identify live variables (to eliminate useless stores), and to calculate reaching definitions (to identify loop invariants; also useful for finding live ranges of virtual registers). We also noted that it can be used for global constant propagation, copy propagation, conversion to SSA form, and a host of other purposes.

An obvious question for both the writers and users of compilers is: among the many possible code improvement techniques, which produce the most "bang for the buck"? For modern machines, particularly those with in-order pipelines, instruction scheduling and register allocation are definitely on the list. Significant additional benefits accrue from some sort of global register allocation, if only to avoid repeated loads and stores of loop indices and other heavily used local variables and parameters. Beyond these basic techniques, which mainly amount to making good use of the hardware, the most significant benefits in von Neumann programs come from optimizing references to arrays, particularly within loops. Most production-quality compilers (1) perform at least enough common subexpression analysis to identify redundant address calculations for arrays, (2) hoist invariant calculations out of loops, and (3) perform strength reduction on induction variables, eliminating them if possible.

As we noted in the introduction to the chapter, code improvement remains an extremely active area of research. Much of this research addresses language features and computational models for which traditional optimization techniques have not been particularly effective. Examples include alias analysis for pointers in C, static resolution of virtual method calls in object-oriented languages (to permit inlining and interprocedural optimization), streamlined communication in message-passing languages, and a variety of issues for functional and logic languages. In some cases, new programming paradigms can change the goals of code improvement. For just-in-time compilation of Java or C# programs, for example, the speed of the code improver may be as important as the speed of the code it produces. In other cases, new sources of information (e.g., feedback from run-time profiling) create new opportunities for improvement. Finally, advances in processor architecture (multiple pipelines, very wide instruction words, out-of-order execution, architecturally visible caches, speculative instructions) continue to create new challenges; processor design and compiler design are deeply interrelated.

17.10 Exercises

17.1 In Section C-17.2 we suggested replacing the instruction r1 := r2 / 2 with the instruction r1 := r2 >> 1, and noted that the replacement may not be correct for negative numbers. Explain the problem. You will want to learn the difference between *logical* and *arithmetic* shift operations (see almost any assembly language manual). You will also want to consider the issue of rounding.

17.2 Prove that the division operation in the loop of the `combinations` subroutine (Example C-17.10) always produces a remainder of zero. Explain the need for the parentheses around the numerator.

17.3 Certain code improvements can sometimes be performed by the programmer, in the source-language program. Examples include introducing additional variables to hold common subexpressions (so that they need not be recomputed), moving invariant computations out of loops, and applying strength reduction to induction variables or to multiplications by powers of two. Describe several optimizations that cannot reasonably be performed by the programmer, and explain why some that could be performed by the programmer might best be left to the compiler.

17.4 In Section 6.5.1, we suggested that the loop

```
// before
for (i = low; i <= high; i++) {
    // during
}
// after
```

be translated as

```
    −− before
    i := low
    goto test
top:
    −− during
    i +:= 1
test:
    if i ≤ high goto top
    −− after
```

And indeed this is the translation we have used for the `combinations` subroutine. The following is an alternative translation:

```
    −− before
    i := low
    if i > high goto bottom
```

```
top:
    –– during
    i +:= 1
    if i ≤ high goto top
bottom:
    –– after
```

Explain why this translation might be preferable to the one we used. (Hints: Consider the number of branches, the migration of loop invariants, and opportunities to fill delay slots.)

17.5 Beginning with the translation of the previous exercise, reapply the code improvements discussed in this chapter to the `combinations` subroutine.

17.6 Give an example in which the numbered heuristics listed under "Dependence Analysis" in Section C-17.6 do not lead to an optimal code schedule.

17.7 Show that forward data flow analysis can be used to verify that a variable is assigned a value on every possible control path leading to a use of that variable (this is the notion of *definite assignment*, described in Section 6.1.3).

17.8 In Sidebar 16.3, we noted two additional properties (other than definite assignment) that a Java Virtual Machine must verify in order to protect itself from potentially erroneous bytecode. On every possible path to a given statement *S* (a) every variable read in *S* must have the same type (which must of course be consistent with operations in *S*), and (b) the operand stack must have the same current depth, and must not overflow or underflow in *S*. Describe how data flow analysis can be used to verify these properties.

17.9 Show that *very busy* expressions (those that are guaranteed to be calculated on every future code path) can be detected via backward, all-paths data flow analysis. Suggest a space-saving code improvement for such expressions.

17.10 Explain how to gather information during local value numbering that will allow us to identify the sets of variables and registers that contributed to the value of each virtual register. (If the value of register v*i* depends on the value of register v*j* or of variable x, then during available expression analysis we say that v*i* $\in Kill_B$ if *B* contains an assignment to v*j* or x and does not contain a subsequent assignment to v*i*.)

17.11 Show how to strength-reduce the expression i^2 within a loop, where i is the loop index variable. You may assume that the loop step size is one.

17.12 Division is often much more expensive than addition and subtraction. Show how to replace expressions of the form i div c on the inside of a `for` loop with additions and/or subtractions, where i is the loop index variable and c is an integer constant. You may assume that the loop step size is one.

17.13 Consider the following high-level pseudocode:

```
read(n)
for i in 1 . . 100
    B[i] := n × i
    if n > 0
        A[i] := B[i]
```

The condition n > 0 is loop invariant. Can we move it out of the loop? If so, explain how. If not, explain why.

17.14 Should live variable analysis be performed before or after loop invariant elimination (or should it be done twice, before *and* after)? Justify your answer.

17.15 Starting with the naive gcd code of Figure 1.7, show the result of local redundancy elimination (via value numbering) and instruction scheduling.

17.16 Continuing the previous exercise, draw the program's control flow graph and show the result of global value numbering. Next, use data flow analysis to drive any appropriate global optimizations. Then draw and color the register conflict graph in order to perform global register allocation. Finally, perform a final pass of instruction scheduling. How does your code compare to the version in Example 1.2?

17.17 In Section C-17.6, we noted that hardware register renaming can often hide anti- and output dependences. Will it help in Figure C-17.13? Explain.

17.18 Consider the following code:

```
v2 := *v1
v1 := v1 + 20
v3 := *v1
—
v4 := v2 + v3
```

Show how to shorten the time required for this code by moving the update of v1 forward into the delay slot of the second load. (Assume that v1 is still live at the end.) Describe the conditions that must hold for this type of transformation to be applied, and the alterations that must be made to individual instructions to maintain correctness.

17.19 Consider the following code:

```
v5 := v2 × v36
—
—
—
—
v6 := v5 + v1
v1 := v1 + 20
```

Show how to shorten the time required for this code by moving the update of v1 backward into a delay slot of the multiply. Describe the conditions that must hold for this type of transformation to be applied, and the alterations that must be made to individual instructions to maintain correctness.

17.20 In the spirit of the previous two exercises, show how to shorten the main loop of the `combinations` subroutine (prior to unrolling or pipelining) by moving the updates of v26 and v34 backward into delay slots. What percentage impact does this change make in the performance of the loop?

17.21 Using the code in Figures C-17.12 and C-17.14 as a guide, unroll the loop of the `combinations` subroutine three times. Construct a dependence DAG for the new Block 2. Finally, schedule the block. How many cycles does your code consume per iteration of the original (unrolled) loop? How does it compare to the software pipelined version of the loop (Figure C-17.16)?

17.22 Write a version of the `combinations` subroutine whose loop is both unrolled *and* software pipelined. In other words, build the loop body from the instructions between the left-most and right-most vertical bars of Figure C-17.15, rather than from the instructions between adjacent bars. You should update the array pointers only once per iteration. How many cycles does your code consume per iteration of the original loop? How messy is the code to "prime" and "flush" the pipeline, and to check for sufficient numbers of iterations?

17.23 Consider the following code for matrix multiplication:

```
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        C[i][j] = 0;
    }
}
for (i = 0; i < n; i++) {
    for (j = 0; j < n; j++) {
        for (k = 0; k < n; k++) {
            C[i][j] += A[i][k] * B[k][j];
        }
    }
}
```

Describe the access patterns for matrices `A`, `B`, and `C`. If the matrices are large, how many times will each cache line be fetched from memory? Tile the inner two loops. Describe the effect on the number of cache misses.

17.24 Consider the following simple instance of Gaussian elimination:

```
for (i = 0; i < n-1; i++) {
    for (j = i+1; j < n; j++) {
        for (k = n-1; k >= i; k--) {
            A[j][k] -= A[i][k] * A[j][i] / A[i][i];
        }
    }
}
```

(Gaussian elimination serves to triangularize a matrix. It is a key step in the solution of systems of linear equations.) What are the loop invariants in this code? What are the loop-carried dependences? Discuss how to optimize the code. Be sure to consider locality-improving loop transformations.

17.25 Modify the tiled matrix transpose of Example C-17.32 to eliminate the `min` calculations in the bounds of the inner loops. Perform the same modification on your answer to Exercise C-17.23.

17.11 Explorations

17.26 Investigate the back-end structure of your favorite compiler. What levels of optimization are available? What techniques are employed at each level? What is the default level? Does the compiler generate assembly language or object code?

Experiment with optimization in several program fragments. Instruct the compiler to generate assembly language, or use a disassembler or debugger to examine the generated object code. Evaluate the quality of this code at various levels of optimization.

If your compiler employs a separate assembler, compare the assembler input to its disassembled output. What optimizations, if any, are performed by the assembler?

17.27 As a general rule, a compiler can apply a program transformation only if it preserves the correctness of the code. In some circumstances, however, the correctness of a transformation may depend on information that will not be known until run time. In this case, a compiler may generate two (or more) versions of some body of code, together with a run-time check that chooses which version to use, or customizes a general, parameterized version.

Learn about the "inspector-executor" compilation paradigm pioneered by Saltz et al. [SMC91]. How general is this technique? Under what circumstances can the performance benefits be expected to outweigh the cost of the run-time check and the potential increase in code size?

17.28 Static compiler analysis can be used to check for patterns of information flow that are likely (though not certain) to constitute programming errors. Investigate the work of Guyer et al. [GL05], which performs analysis reminiscent of *taint mode* (Exploration 16.21) at compile time. In a similar vein, investigate the work of Yang et al. [YTEM04] and Chen et al. [CDW04], which use static *model checking* to catch high-level errors. What do you think of such efforts? How do they compare to taint mode or to *proof-carrying code* (Exploration 16.22)? Can static analysis be useful if it has both false negatives (errors it misses) and false positives (correct code it flags as erroneous)?

17.29 In a somewhat gloomy parody of Moore's Law, Todd Proebsting (an eminent compiler researcher formerly at Microsoft Research and now on the faculty of the University of Arizona) once coined what he called *Proebsting's Law*: "Compiler advances double computing power every 18 years."

Survey the history of compiler technology. What have been the major innovations? Have there been important advances in areas other than speed? Is Proebsting's Law a fair assessment of the field?

17.12 Bibliographic Notes

Mainstream compiler textbooks (e.g., those of Cooper and Torczon [CT11], Grune et al. [GBJ$^+$12], or Aho et al. [ALSU07]) are an accessible source of information on back-end compiler technology. Much of the presentation here was inspired by Muchnick's *Advanced Compiler Design and Implementation*, which contains a wealth of detailed information and citations to related work [Muc97]. Much of the leading-edge compiler research appears in the annual *ACM Conference on Programming Language Design and Implementation* (PLDI). A compendium of "best papers" from the first 20 years of this conference was published in 2004 [McK04].

Throughout our study of code improvement, we concentrated our attention on the von Neumann family of languages. Analogous techniques for functional [App91; Pey87; Pey92; WM95, Chap. 3; App97, Chap. 15; GBJ$^+$12, Chap. 7]; object-oriented [AH95; GDDC97; WM95, Chap. 5; App97, Chap. 14; GBJ$^+$12, Chap. 6]; and logic languages [DRSS96; FSS83; Zho96; WM95, Chap. 4; GBJ$^+$12, Chap. 8] are an active topic of research, but are beyond the scope of this book. A key challenge in functional languages is to identify repetitive patterns of calls (e.g., tail recursion), for which loop-like optimizations can be performed. A key challenge in object-oriented languages is to predict the targets of virtual subroutine calls statically, to permit in-lining and interprocedural code improvement. The dominant challenge in logic languages is to better direct the underlying process of goal-directed search.

Local value numbering is originally due to Cocke and Schwartz [CS69]; the global algorithm described here is based on that of Alpern, Wegman, and Zadeck [AWZ88]. Chaitin et al. [CAC$^+$81] popularized the use of graph coloring for register allocation. Cytron et al. [CFR$^+$91] describe the generation and use of static single-assignment form. Allen and Kennedy [AK02, Sec. 12.2] discuss the general problem of alias analysis in C. Pointers have historically been the most difficult part of this analysis; Smaragdakis and Balatsouras [SB15] provide a tutorial survey. Instruction scheduling from basic-block dependence DAGs is described by Gibbons and Muchnick [GM86]. The general technique is known as *list scheduling*; explanations appear in the texts of Muchnick [Muc97, Sec. 17.1.2] and Cooper and Torczon [CT11, Sec. 12.3]. Massalin provides a delightful discussion of circumstances under which it may be desirable (and possible) to generate a truly *optimal* program [Mas87]. Several projects have expanded on this idea; see for example the work of Schkufza et al. [SSA13].

Sources of information on loop transformations and parallelization include the text of Allen and Kennedy [AK02], the classic text of Wolfe [Wol96], and the excellent survey of Bacon, Graham, and Sharp [BGS94]. Banerjee provides a detailed discussion of loop dependence analysis [Ban97]. Rau and Fisher discuss fine-grain *instruction-level* parallelism, of the sort exploitable by vector, wide-instruction-word, or superscalar processors [RF93].