2 Programming Language Syntax

2.3.4 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 Modula-2:

```
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. 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 that follow the erroneous expression.

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 (input_token $\notin$ FIRST($foo$)) and (not EPS($foo$))
        report_error                  –– print message for the user
        repeat
            delete_token
        until input_token $\in$ (FIRST($foo$) $\cup$ FOLLOW($foo$) $\cup$ {$$})
    case input_token of
        . . . : . . .
        . . . : . . .                 –– valid starting tokens
        . . . : . . .
        otherwise return              –– error or $foo \longrightarrow \epsilon$
```

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(symbol)
    if (input_token $\notin$ FIRST(symbol)) and (not EPS(symbol))
        report_error
        repeat
            delete_token
        until input_token $\in$ (FIRST(symbol) $\cup$ FOLLOW(symbol) $\cup$ {$$})
```

■

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, as mentioned in Section 2.3.3 [page 91]).

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* $\longrightarrow$ `id :=` *expr*, but *not* into the second, inner expr, which is called as part of the production *factor* $\longrightarrow$ `(` *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(symbol, follow_set)
    if (input_token $\notin$ FIRST(symbol)) and (not EPS(symbol))
        report_error
        repeat
            delete_token
        until input_token $\in$ FIRST(symbol) $\cup$ follow_set $\cup$ starter_set $\cup$ {$$}
```

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

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 8.5). 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 ∈ FIRST(statement)
                statement          –– try again
                return
            elsif next_token ∈ FOLLOW(statement)
                return
            else get_next_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, FL88]. 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]

The FMQ error-repair algorithm requires the compiler writer to assign an insertion cost $C(\texttt{t})$ and a deletion cost $D(\texttt{t})$ to every token `t`. (Since we cannot change where the input ends, we have $C(\texttt{\$\$}) = D(\texttt{\$\$}) = \infty$.) In any given error situation, the algorithm chooses the least cost combination of insertions and

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 $A \longrightarrow \epsilon$, the FIRST set of the third symbol B from the top or, if it would cause us to predict $B \longrightarrow \epsilon$, the FIRST set of the fourth symbol from the top, and so on.

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.

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,

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

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 $X = \mathtt{a}$, or if $X \Longrightarrow^* \mathtt{a}x$, then $E(X, \mathtt{a}) = \epsilon$, where $C(\epsilon) = 0$.

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 ◎2.30. 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 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

```
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 2.30 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 2.31 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.

the obvious way. We then embed calls to find_insertion in a second loop, shown in Figure ◎2.31. 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 re-enables the reporting of errors after successfully shifting three real tokens of input. A semantic action routine can re-enable 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 2.24. 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, 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");}
```

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.

In the CFSM of Figure 2.25, 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

45. Why is syntax error recovery important?
46. What are *cascading errors*?
47. What is *panic mode*? What is its principal weakness?
48. What is the advantage of *phrase-level recovery* over panic mode?
49. What is the *immediate error detection problem*, and how can it be addressed?
50. Describe two situations in which context-specific FOLLOW sets may be useful.
51. Outline Wirth's mechanism for error recovery in recursive descent parsers. Compare this mechanism to exception-based recovery.
52. What are *error productions*? Why might a parser that incorporates a high-quality, general-purpose error recovery algorithm still benefit from using such productions?
53. Outline the FMQ algorithm. In what sense is the algorithm optimal?
54. Why is error recovery more difficult in bottom-up parsers than it is in top-down parsers?
55. 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 (page 56) 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* $\delta : Q \times \Sigma \to Q$ 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, to be the set $\{x \mid \delta(q_1, x) \in F\}$. In a nondeterministic finite automaton (NFA),

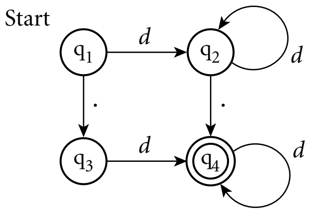


Figure 2.32 **Minimal DFA for the language consisting of all strings of decimal digits containing a single decimal point.** Adapted from Figure 2.10 in the main text. The symbol d here is short for "0, 1, 2, 3, 4, 5, 6, 7, 8, 9".

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 2.32 (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 2.32). More formal and general treatment of all the regular language constructions can be found in standard automata theory texts [HMU01, Sip97].

From a DFA to a Regular Expression

To construct a regular expression equivalent to a given DFA, we employ a 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

$$r_{ii}^0 = \texttt{a}_1 \mid \texttt{a}_2 \mid \ldots \mid \texttt{a}_m \mid \epsilon$$

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, $r_{ij}^0 = \epsilon$. Similarly, for $i \neq j$, we define

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

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_{ij}^0 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_{ii}^0, but not in r_{ij}^0 when $i \neq j$.)

Given these r^0 expressions, the dynamic programming algorithm inductively calculates expressions r_{ij}^k 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, $r_{11}^0 = r_{33}^0 = \epsilon$, and $r_{22}^0 = r_{44}^0 = \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} \mid \epsilon$, which we will abbreviate $d \mid \epsilon$. Similarly, $r_{13}^0 = r_{24}^0 = \texttt{.}$, and $r_{12}^0 = r_{34}^0 = d$. Expressions r_{14}^0, r_{21}^0, r_{23}^0, r_{31}^0, r_{32}^0, r_{41}^0, r_{42}^0, and r_{43}^0 are all empty.

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

$$r_{ij}^k = r_{ij}^{k-1} \mid r_{ik}^{k-1} r_{kk}^{k-1\,\star} r_{kj}^{k-1}$$

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 $r_{1f_1}^n \mid r_{1f_2}^n \mid \ldots \mid r_{1f_t}^n$, 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 $r_{11}^0 = \epsilon$ 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, $\forall i\,[\,r_{ii}^1 = r_{ii}^0\,]$. The second step is a bit more interesting. Since we are now allowed to go through State 2, we have $r_{22}^2 = r_{22}^2\ r_{22}^{2\,\star}\, r_{22}^2 = (\,d \mid \epsilon\,) \mid (\,d \mid \epsilon\,) \mid (\,d \mid \epsilon\,)^\star \mid (\,d \mid \epsilon\,) \mid = d^\star$. Because r_{21}^1, r_{23}^1, r_{32}^1, and r_{42}^1 are empty, however, r_{11}^2, r_{33}^2, and r_{44}^2 remain the same as r_{11}^1, r_{33}^1, and r_{44}^1. In a similar vein, we have

$$r_{12}^2 = d \mid d\,(\,d \mid \epsilon\,)^\star(\,d \mid \epsilon\,) = d^+$$
$$r_{14}^2 = d\,(\,d \mid \epsilon\,)^\star\ \texttt{.} = d^+\ \texttt{.}$$
$$r_{24}^2 = \texttt{.}\ \mid (\,d \mid \epsilon\,)\,(\,d \mid \epsilon\,)^\star\ \texttt{.} = d^\star\ \texttt{.}$$

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

$$\begin{aligned} r_{13}^3 &= . \mid . \, \epsilon^\star \epsilon = . \\ r_{14}^3 &= d^+ . \mid . \, \epsilon^\star d = d^+ . \mid . \, d \\ r_{34}^3 &= d \mid \epsilon\epsilon^\star d = d \end{aligned}$$

All other expressions remain unchanged from the previous step.

Finally, we have

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

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 ◎2.32 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 ◎2.33, 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 ◎2.34. 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 | b | c )* b ( a | b | c )* b ( a | b | c )* | ( a | b | c )* c ( a | b | c )* c ( a | b | c )*`. ■

Of course, the eight-state NFA does not emerge directly from the construction of Section 2.2.1; that construction produces a 52-state machine with a certain amount of redundancy, and with many extraneous states and epsilon productions.

EXAMPLE 2.59
Exponential DFA blow-up

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

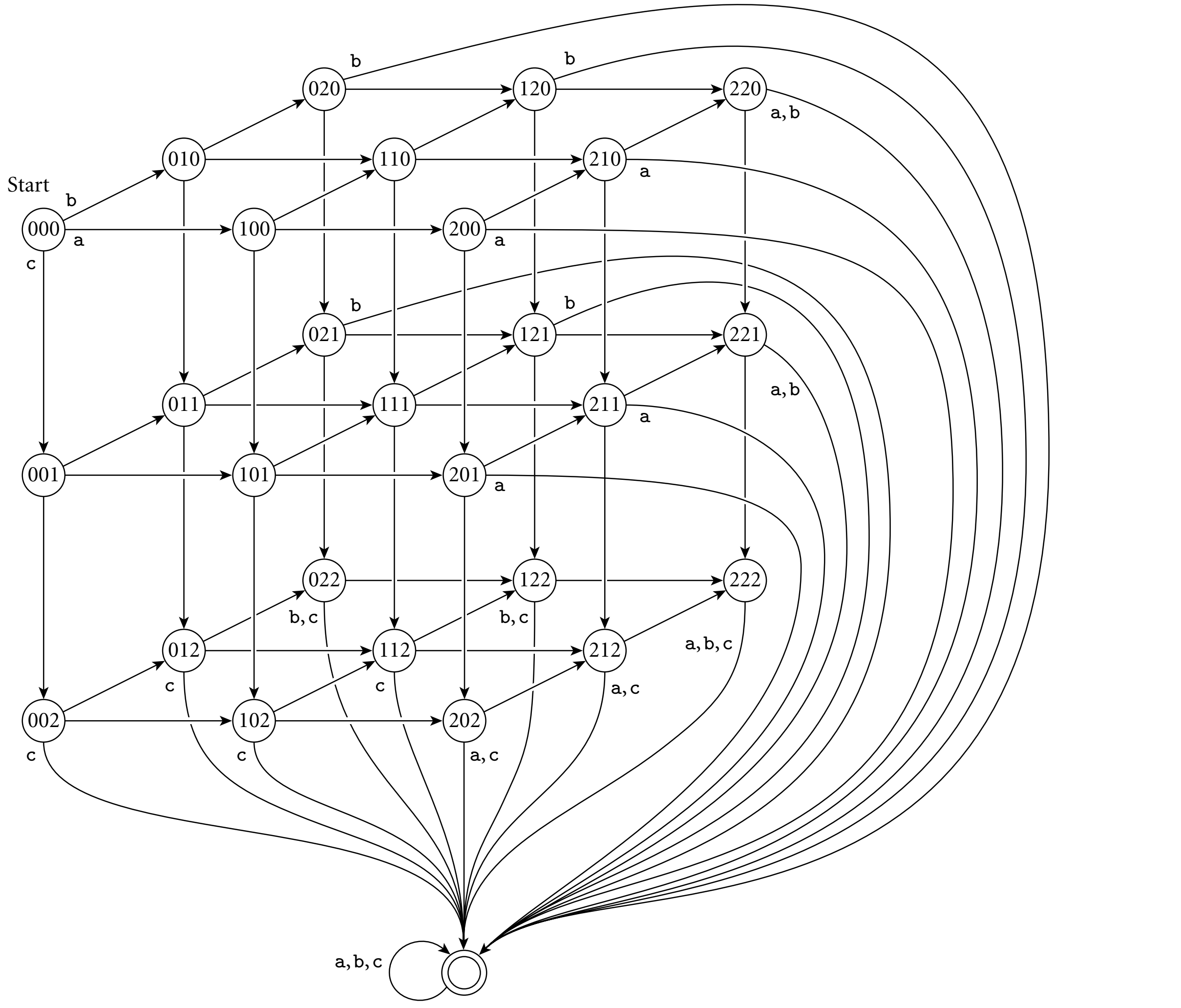


Figure 2.33 DFA for the language consisting of all strings in `( a | b | c )*` in which some letter appears at least three times. State name *ijk* indicates that *i* `a`s, *j* `b`s, and *k* `c`s have been seen so far. Within the cubic portion of the figure, most edge labels are elided: `a` transitions 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)*)
|  ...
| ((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)*)
```

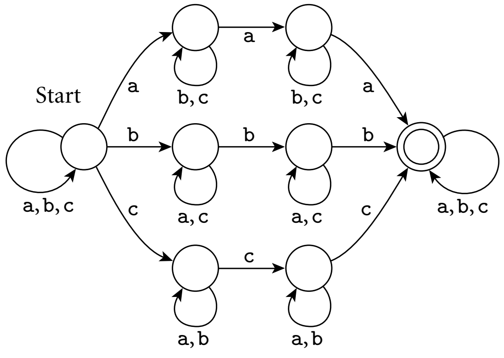


Figure 2.34 NFA corresponding to the DFA of Figure 2.33.

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 $\delta : Q \times \Gamma \times \{\Sigma \cup \{\epsilon\}\} \rightarrow Q \times \Gamma^*$, 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 $\delta(q,\epsilon, Z)$ is defined, N moves to state r and replaces Z with α in the stack, where $(r, \alpha) = \delta(q,\epsilon, Z)$. In this case N does not consume its input symbol. If $\delta(q,\epsilon, Z)$ is undefined, N examines and consumes the current input symbol a. It then moves to state s and replaces Z with β, where $(s, \beta) = \delta(q, \texttt{a}, Z)$. 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 $\delta(q,\epsilon, Z)$ 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, 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 2.3.3 (page 91), 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 2.3.2 and 2.3.3.

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 FOLLOWsets 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 Ⓒ2.3.4. 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 2.3.2, 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 Ⓒ2.35. 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 Ⓒ2.36. Proofs that they lie in the regions claimed are deferred to Exercise Ⓒ2.30. ■

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). Containment relationships among the classes of languages accepted by the popular parsing algorithms appear in Figure Ⓒ2.37. Again, languages can be found in every region. Examples appear in Figure Ⓒ2.38; proofs are deferred to Exercise Ⓒ2.31. ■

EXAMPLE 2.62
Separation of language classes

Note that every context-free language that can be parsed deterministically has an SLR(1) grammar. Moreover, any language that can be parsed deterministically and in which no valid string can be extended to create another valid string (this is called the *prefix property*) has an LR(0) grammar. If we restrict our attention to languages with an explicit `$$` marker at end-of-file, then they all have the prefix property, and therefore LR(0) grammars.

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

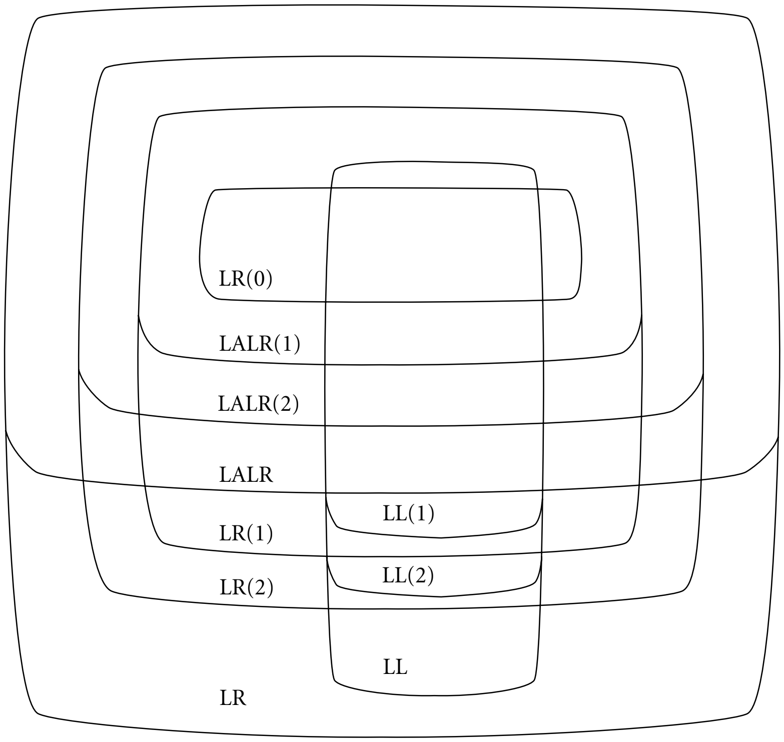


Figure 2.35 **Containment relationships among popular grammar classes.** In addition to the containments shown, SLL(k) is just inside LL(k), for $k \geq 2$, but has the same relationship to everything else, and SLR(k) is just inside LALR(k), for $k \geq 1$, but has the same relationship to everything else.

LL(2) but not SLL:

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

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

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

LR(0) but not LL:

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

SLL(1) but not LALR:

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

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

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

LALR(1) but not SLR:

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

LR(1) but not LALR:

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

Unambiguous but not LR:

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

Figure 2.36 Examples of grammars in various regions of Figure ◎2.35.

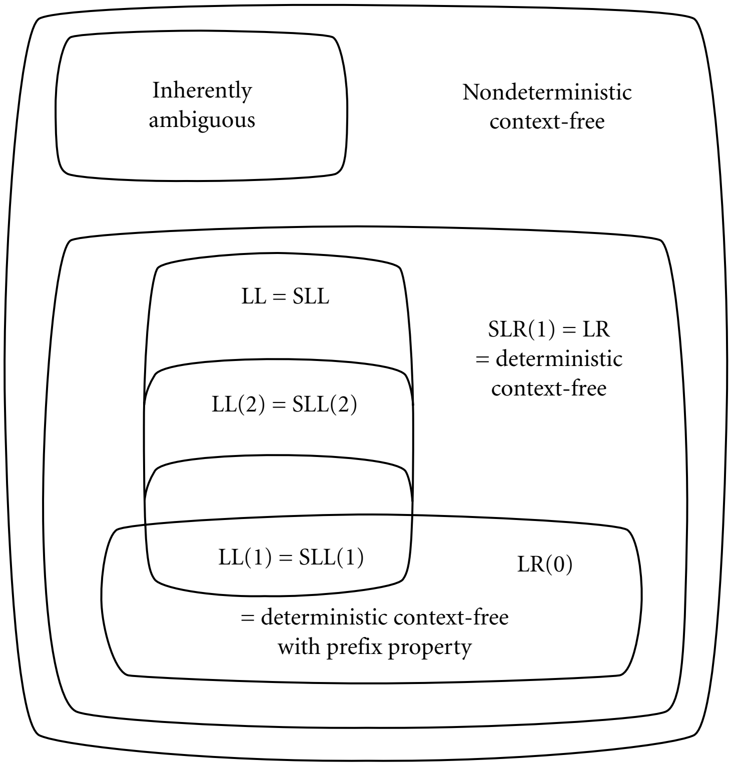

Inherently ambiguous
Nondeterministic context-free
LL = SLL
SLR(1) = LR = deterministic context-free
LL(2) = SLL(2)
LL(1) = SLL(1)
LR(0)
= deterministic context-free with prefix property

Figure 2.37 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:
$\{\texttt{a}^n\texttt{(} \texttt{b} \mid \texttt{c} \mid \texttt{b}^k\texttt{d} \texttt{)}^n : n \geq 1\}$
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 2.38 Examples of languages in various regions of Figure 2.37.

✓ CHECK YOUR UNDERSTANDING

56. What formal machine captures the behavior of a scanner? A parser?
57. State three ways in which a real scanner differs from the formal machine.

58. What are the formal components of a DFA?
59. Outline the algorithm used to construct a regular expression equivalent to a given DFA.
60. What is the inherent "big-O" complexity of parsing with an NPDA? Why is this worse than the $O(n^3)$ time mentioned in Section 2.3?
61. How many states are there in an LL(1) PDA? An SLR(1) PDA? Explain.
62. What are the *viable prefixes* of a CFG?
63. Summarize the proof that a DFA cannot recognize arbitrarily nested constructs.
64. Explain the difference between LL and SLL parsing.
65. Is every LL(1) grammar also LR(1)? Is it LALR(1)?
66. Does every LR language have an SLR(1) grammar?
67. Why are the containment relationships among grammar classes more complex than those among language classes?

2 Programming Language Syntax

2.6 Exercises

2.28 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.

2.29 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).

2.30 Prove that the grammars in Figure 2.36 lie in the regions claimed.

2.31 (Difficult) Prove that the languages in Figure 2.38 lie in the regions claimed.

2.32 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

2.40 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.

2.41 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 Pascal and other languages with `with` statements (Section 7.3.3), field names must become visible in a multi-statement context.

- 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. 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.
- 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*. The debugger is a tool that 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 from the user (e.g., to print the value of `my_firm^.revenues[1999]`), the debugger must have access to the compiler's symbol table. To make it available at run time, the compiler typically saves the table in a hidden portion of the final machine-language program.

EXAMPLE 3.44
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.

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
```

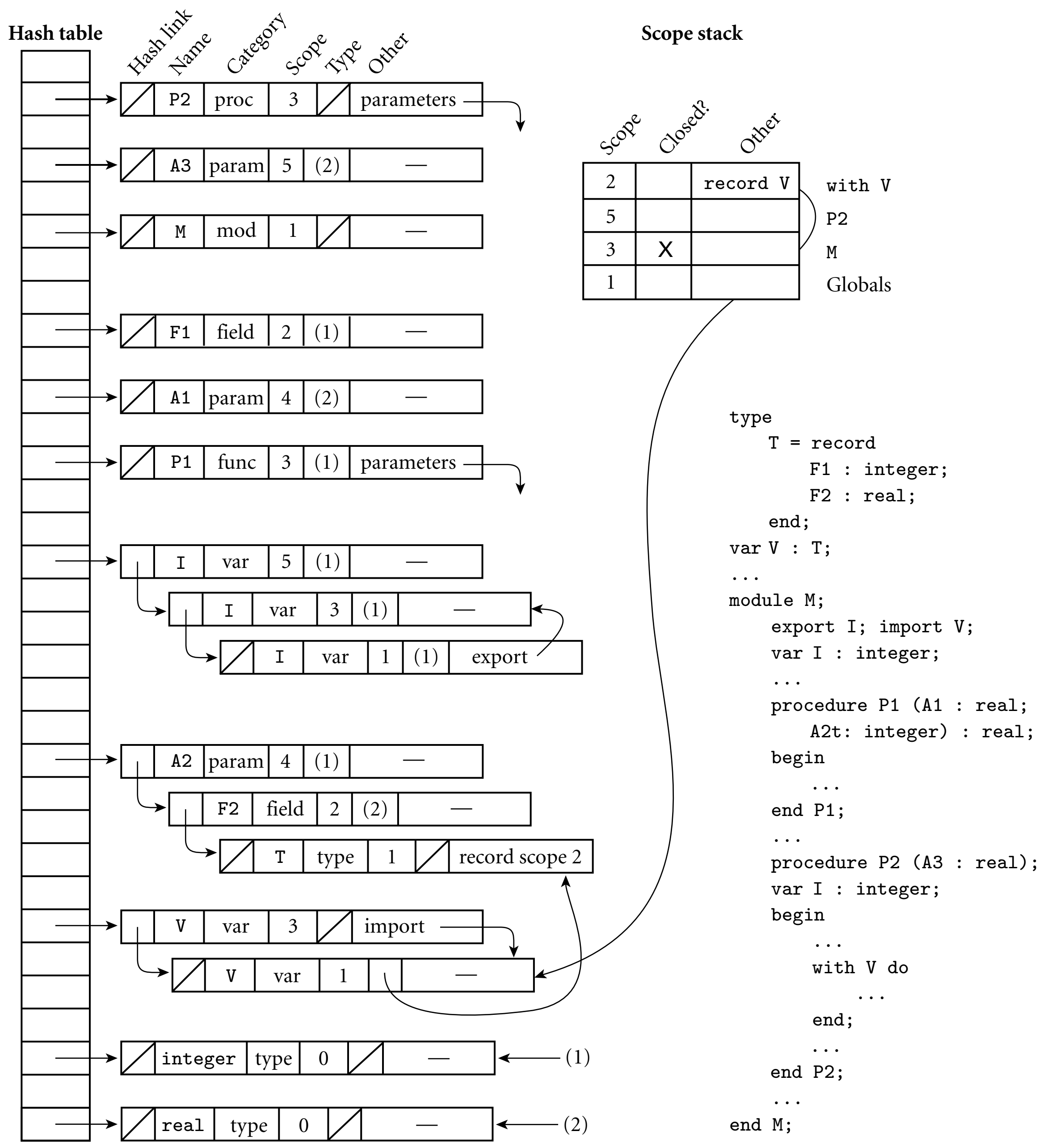
Figure 3.18 LeBlanc-Cook symbol table lookup operation.

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 ◎3.18. ■

EXAMPLE 3.45
Symbol table for a sample program

The lower right portion of Figure ◎3.19 contains the skeleton of a Modula-2 program. The remainder of the figure shows the configuration of the symbol table for the referencing environment of the `with` statement in procedure `P2`. The scope stack contains four entries representing, respectively, the `with` statement, procedure `P2`, module `M`, and the global scope. The scope for the `with` statement indicates the specific record variable 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, `A2`, `F2`, and `T` have also ended up on a single chain, due to hash collisions. Variables `V` and `I` (`M`'s `I`) have extra entries, to make them visible across the boundary of closed scope `M`. When we are inside



```
type
   T = record
      F1 : integer;
      F2 : real;
   end;
var V : T;
...
module M;
   export I; import V;
   var I : integer;
   ...
   procedure P1 (A1 : real;
      A2t: integer) : real;
   begin
      ...
   end P1;
   ...
   procedure P2 (A3 : real);
   var I : integer;
   begin
      ...
      with V do
         ...
      end;
      ...
   end P2;
   ...
end M;
```

Figure 3.19 LeBlanc-Cook symbol table for an example program in a language like Modula-2. The scope stack represents the referencing environment of the `with` statement in procedure `P2`. For the sake of clarity, the many pointers from type fields to the symbol table entries for `integer` and `real` are shown as parenthesized (1)s and (2)s, rather than as arrows.

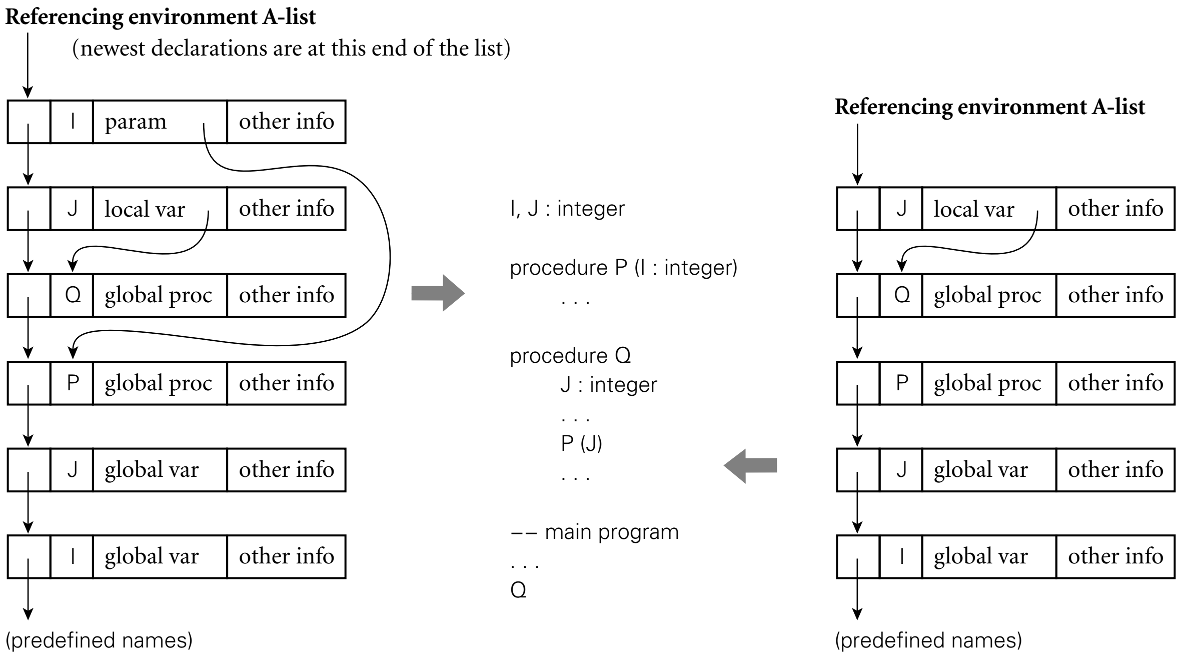


Figure 3.20 **Dynamic scoping with an association list.** The left side of the figure shows the referencing environment at the point in the code indicated by the adjacent grey arrow: after the main program calls Q and it in turn calls P. When searching for I, one will find the parameter at the beginning 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.

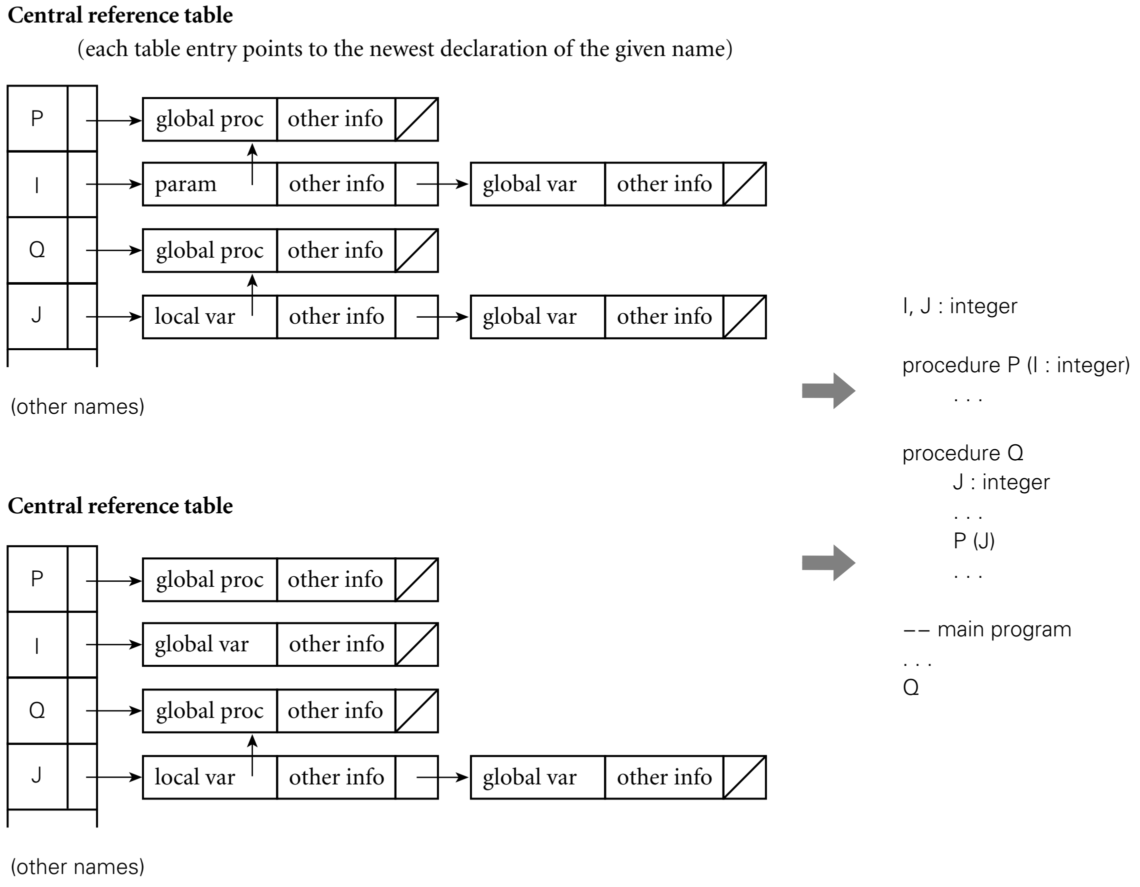
P2, a lookup operation on I will find P2's I; neither of the entries for M's I will be visible. The entry for type T indicates the scope number to be pushed onto the scope stack during with statements. The entry for each subroutine contains the head pointer of a list that links together the subroutine's 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. ■

3.4.2 Association Lists and Central Reference Tables

Pictorial representations of the two principal implementations of dynamic scoping appear in Figures ◎3.20 and ◎3.21. Association 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 3.46
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



```
I, J : integer

procedure P (I : integer)
      . . .

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

-- main program
. . .
Q
```

Figure 3.21 **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.

for Lisp interpreters 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 ◎3.20, 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 3.47
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 ◎3.21, 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 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.

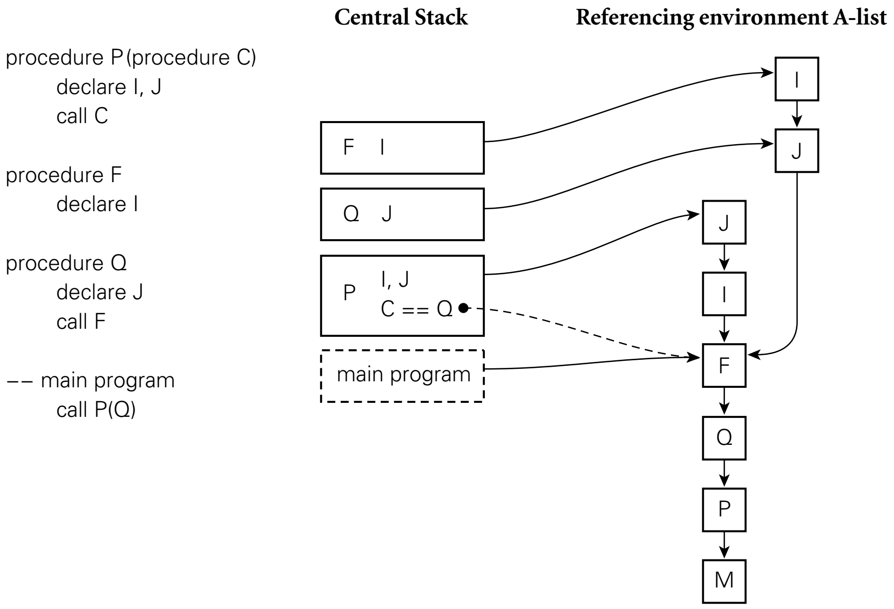
Within the Lisp community, implementation of dynamic scoping via an association list is sometimes called *deep binding*, because the lookup operation may need to look arbitrarily deep in the list. Implementation via a central reference table is sometimes called *shallow binding*, because it finds the 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" [Seb08] 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 3.48
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 ◎3.22). When 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



```
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 3.22 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.

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 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. ■

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**

44. List the basic operations provided by a symbol table.
45. Outline the implementation of a LeBlanc-Cook style symbol table.
46. Why don't compilers generally remove names from the symbol table at the ends of their scopes?
47. Describe the *association list* (*A-list*) and *central reference table* data structures used to implement dynamic scoping. Summarize the tradeoffs between them.
48. 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 14 and 15.

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 3.49
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 3.50
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. Note that linkage remains the foundation for separate compilation: .h files are merely a convention. ■

3.8.2 Packages and Automatic Header Inference

EXAMPLE 3.51
Packages in Java

The separate compilation facilities of Java and C# eliminate .h files. Specifically, 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 foo_type_1 { ...
```

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 3.52
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.foo_type_1 my_first_obj;
```

or

```
import foo.foo_type_1;
...
foo_type_1 my_first_obj;
```

or

```
import foo.*;           // import everything from foo
...
foo_type_1 my_first_obj;
```
■

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 3.53
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. We will consider the Ada 95 facilities further in Section 9.2.4. ■

✓ CHECK YOUR UNDERSTANDING

49. What purpose(s) does separate compilation serve?
50. What does it mean for an external variable to be *linked* in C?
51. Summarize the C conventions for use of .h and .c files.
52. Describe the difference between a compilation unit and a C++ or C# *namespace.*
53. 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

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 common block). (And as we noted in Section 3.3.1 [page 123], the lack of type checking for common blocks was originally considered a feature, not a bug!)

3 Names, Scopes, and Bindings

3.10 Exercises

3.23 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]`.

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

(a) function `F1` in Figure 3.4 (page 126).

(b) procedure pop in Figure 3.7 (page 136).

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

(a) Figure 3.9 (page 140). Assume that a positive value is read at line 8.

(b) Exercise 3.14.

3.26 Repeat the previous exercise for a central reference table.

3.27 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.

3.28 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).

3.29 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 9.2.)

3 Names, Scopes, and Bindings

3.11 Explorations

3.39 Learn about the `.stabs` directives used by Unix compilers and assemblers to include symbol table information in object and executable files, where it will be accessible to symbolic debuggers. Write a brief tutorial that might help a systems programmer understand the directives found in assembly language files.

3.40 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 15.3.1.

3.41 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 the sidebar on page 707 for some initial pointers.)

3.42 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?

3.43 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 Semantic Analysis

4.5 Space Management for Attributes

A compiler that does not build an explicit parse tree requires some other mechanism to allocate, deallocate, and refer to storage space for attributes. In the two subsections below we consider attribute space management for bottom-up and top-down parsers, respectively. For bottom-up parsers the principal challenge is where to put the inherited attributes of symbols that have not yet been seen, and thus have no record in the parse stack. For top-down parsers this challenge does not arise, but we must go to a bit more effort to retain space for symbols that have already been parsed, and we must choose whether to manage this space automatically or to give some of the burden to the writer of action routines.

4.5.1 Bottom-Up Evaluation

EXAMPLE 4.19
Stack trace for bottom-up parse, with action routines

Figure ©4.17 shows a trace of the parse and attribute stack for `(1 + 3) * 2`, using the attribute grammar of Figure 4.1. 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 4.1 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 `$$`. Attributes of right-hand side symbols would be accessed as fields of the pseudostructs `$1`, `$2`, etc. To get from line 9 to line 10, for example, in the trace of Figure ©4.17, we would use an action routine version of the first rule of the grammar in Figure 4.1: $$.val = $1.val + $3.val. ■

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 4.17 Parse/attribute stack trace for `(1 + 3) * 2`, using the grammar of Figure 4.1. Subscripts represent val attributes; they are not meant to distinguish among instances of a symbol.

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.

Inherited Attributes

EXAMPLE 4.20
Finding inherited attributes in "buried" records

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:

$$\begin{aligned}
&dec \longrightarrow type\ \ id\_list\\
&id\_list \longrightarrow \texttt{id}\\
&id\_list \longrightarrow id\_list\ \texttt{,}\ \texttt{id}
\end{aligned}$$

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 4.21
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:

```
stmt ⟶  id := expr
     ⟶  ...
     ⟶  if expr then stmt
expr ⟶  ...
```

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. ■

Semantic Hooks

EXAMPLE 4.22
Semantic hooks for context

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.

$stmt \longrightarrow$ `id := ` A $expr$
$\longrightarrow$ ...
$\longrightarrow$ `if` B $expr$ `then` $stmt$
$A \longrightarrow \epsilon$ { \$\$.tp := \$–1.tp }
$B \longrightarrow \epsilon$ { \$\$.tp := Boolean }
$expr \longrightarrow$ ...{ if \$0.tp = Boolean then ...} ■

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 4.23
Semantic hooks that break an LR CFG

1. $stmt \longrightarrow l\_val$ `:=` $expr$
2. $\longrightarrow$ `id` $args$
3. $l\_val \longrightarrow$ `id` $quals$
4. $quals \longrightarrow quals$ `.` `id`
5. $\longrightarrow quals$ `(` $expr\_list$ `)`
6. $\longrightarrow \epsilon$
7. $args \longrightarrow$ `(` $expr\_list$ `)`
8. $\longrightarrow \epsilon$

An *l-value* in this grammar is a "qualified" identifier: an identifier followed by optional array subscript and record field qualifiers.[1] 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:

$stmt \longrightarrow$ `id` A $args$
$A \longrightarrow \epsilon$ { \$\$.proc_index := lookup (\$0.name) }

1 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.

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 `(`. ■

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.

EXAMPLE 4.24 Action routines in the trailing part

`Yacc/bison` recognizes this fact explicitly by allowing action routines to be embedded in right-hand sides. It automatically converts the production

$S \longrightarrow \alpha$ { your code here } β

to

$S \longrightarrow \alpha\ A\ \beta$
$A \longrightarrow \epsilon$ { your code here }

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 4.25 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:

```
stmt ⟶ id A quals assign_opt
A ⟶ ε { $$.id_index := lookup ($0.name) }
quals ⟶ quals . id
      ⟶ quals ( expr_list )
      ⟶ ε
assign_opt ⟶ := expr
           ⟶ ε
```

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. ■

4.5.2 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 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 4.26
Operation of an LL attribute stack

Figure ©4.18 contains an LL(1) grammar for constant expressions, with action routines. Figure ©4.19 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 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

$$\begin{array}{l}
E \longrightarrow T \; \{ \; \text{TT.st} := \text{T.val} \; \}^{\mathbf{1}} \; TT \; \{ \; \text{E.val} := \text{TT.val} \; \}^{\mathbf{2}} \\
TT_1 \longrightarrow \texttt{+} \; T \; \{ \; \text{TT}_2\text{.st} := \text{TT}_1\text{.st} + \text{T.val} \; \}^{\mathbf{3}} \; TT_2 \; \{ \; \text{TT}_1\text{.val} := \text{TT}_2\text{.val} \; \}^{\mathbf{4}} \\
TT_1 \longrightarrow \texttt{-} \; T \; \{ \; \text{TT}_2\text{.st} := \text{TT}_1\text{.st} - \text{T.val} \; \}^{\mathbf{5}} \; TT_2 \; \{ \; \text{TT}_1\text{.val} := \text{TT}_2\text{.val} \; \}^{\mathbf{6}} \\
TT \longrightarrow \epsilon \, \{ \; \text{TT.val} := \text{TT.st} \; \}^{\mathbf{7}} \\
T \longrightarrow F \; \{ \; \text{FT.st} := \text{F.val} \; \}^{\mathbf{8}} \; FT \; \{ \; \text{T.val} := \text{FT.val} \; \}^{\mathbf{9}} \\
FT_1 \longrightarrow \texttt{*} \; F \; \{ \; \text{FT}_2\text{.st} := \text{FT}_1\text{.st} \times \text{F.val} \; \}^{\mathbf{10}} \; FT_2 \; \{ \; \text{FT}_1\text{.val} := \text{FT}_2\text{.val} \; \}^{\mathbf{11}} \\
FT_1 \longrightarrow \texttt{/} \; F \; \{ \; \text{FT}_2\text{.st} := \text{FT}_1\text{.st} \div \text{F.val} \; \}^{\mathbf{12}} \; FT_2 \; \{ \; \text{FT}_1\text{.val} := \text{FT}_2\text{.val} \; \}^{\mathbf{13}} \\
FT \longrightarrow \epsilon \, \{ \; \text{FT.val} := \text{FT.st} \; \}^{\mathbf{14}} \\
F_1 \longrightarrow \texttt{-} \; F_2 \; \{ \; \text{F}_1\text{.val} := - \; \text{F}_2\text{.val} \; \}^{\mathbf{15}} \\
F \longrightarrow \texttt{(} \; E \; \texttt{)} \; \{ \; \text{F.val} := \text{E.val} \; \}^{\mathbf{16}} \\
F \longrightarrow \texttt{const} \; \{ \; \text{F.val} := \text{C.val} \; \}^{\mathbf{17}}
\end{array}$$

Figure 4.18 LL(1) grammar for constant expressions, with action routines. The boldface superscripts are for reference in Figure ◎4.19.

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 ◎4.20 identifies these symbols graphically at the point in Figure ◎4.19 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.

At start-up, the attribute stack contains a record for the goal 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`. ■

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 ◎4.18. In comparing Figures ◎4.17

`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:$`	$E_?\; T_?\; TT_{?,?}\; F_4\; \boxed{L}\; FT_{4,?}\; \boxed{R}\; \boxed{N}\; \ast\; F_?\; FT_{?,?}$
`F10FT11:9:1TT2:$`	$E_?\; T_?\; TT_{?,?}\; F_4\; \boxed{L}\; FT_{4,?}\; \boxed{R}\; \ast\; \boxed{N}\; F_?\; FT_{?,?}$
`C17:10FT11:9:1TT2:$`	$E_?\; T_?\; TT_{?,?}\; F_4\; FT_{4,?}\; \ast\; \boxed{L}\; F_?\; FT_{?,?}\; \boxed{R}\; \boxed{N}\; C_2$
`17:10FT11:9:1TT2:$`	$E_?\; T_?\; TT_{?,?}\; F_4\; FT_{4,?}\; \ast\; \boxed{L}\; F_?\; FT_{?,?}\; \boxed{R}\; C_2\; \boxed{N}$
`:10FT11:9:1TT2:$`	$E_?\; T_?\; TT_{?,?}\; F_4\; FT_{4,?}\; \ast\; \boxed{L}\; F_2\; FT_{?,?}\; \boxed{R}\; C_2\; \boxed{N}$
`10FT11:9:1TT2:$`	$E_?\; T_?\; TT_{?,?}\; F_4\; \boxed{L}\; FT_{4,?}\; \boxed{R}\; \ast\; F_2\; \boxed{N}\; FT_{?,?}$
`FT11:9:1TT2:$`	$E_?\; T_?\; TT_{?,?}\; F_4\; \boxed{L}\; FT_{4,?}\; \boxed{R}\; \ast\; F_2\; \boxed{N}\; FT_{8,?}$
⟨ 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 4.19 **Trace of the parse stack (left) and attribute stack (right) for `(1 + 3) * 2`, using the grammar (and action routine numbers) of Figure ◎4.18.** Subscripts in the attribute stack indicate the values of attributes. For symbols with two attributes, st comes first.

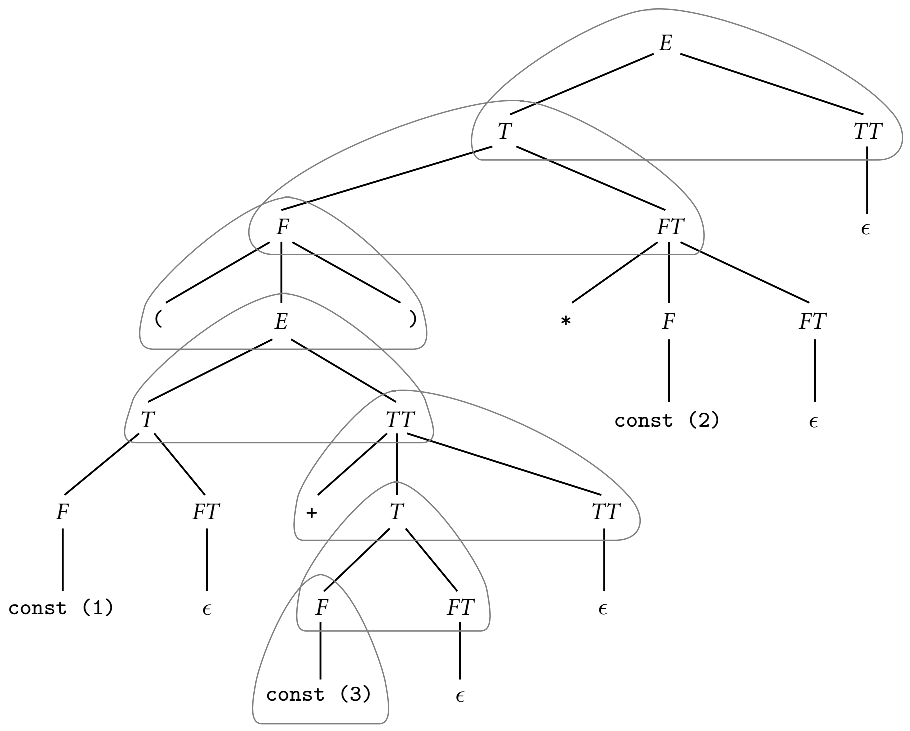


Figure 4.20 Productions with symbols currently in the attribute stack during a parse of `(1 + 3) * 2` (using the grammar of Figure ◎4.18), at the point where we are about to parse the `3`. In Figure ◎4.19, this point corresponds to the line immediately above the eight elided lines.

and ◎4.19, 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 4.9 or ◎4.18, 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 4.27
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 4.9 with the simpler version shown in Figure ◎4.21. 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

$E \longrightarrow T\ TT$

$TT \longrightarrow$ `+` T { bin_op ("+") } TT

$TT \longrightarrow$ `-` T { bin_op ("−") } TT

$TT \longrightarrow \epsilon$

$T \longrightarrow F\ FT$

$FT \longrightarrow$ `*` F { bin_op ("×") } FT

$FT \longrightarrow$ `/` F { bin_op ("÷") } FT

$FT \longrightarrow \epsilon$

$F \longrightarrow$ `-` F { un_op ("$^{+}/_{-}$") }

$F \longrightarrow$ `(` E `)`

$F \longrightarrow$ `const` { push_leaf (cur_tok.val) }

Figure 4.21 Ad hoc management of attribute space in an LL(1) grammar to build a syntax tree.

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.

EXAMPLE 4.28 Processing lists with an attribute stack

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 Section 4.5.1 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:

$dec \longrightarrow id\_list$ `:` $type$

$id\_list \longrightarrow$ `id` id_list_tail

$id\_list\_tail \longrightarrow$ `,` id_list

$\qquad\qquad \longrightarrow \epsilon$

Without resorting to non-L-attributed flow (see Exercise 4.26), 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 4.29
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 mainly a matter of taste.

✓ CHECK YOUR UNDERSTANDING

17. Explain how to manage space for synthesized attributes in a bottom-up parser.
18. Explain how to manage space for inherited attributes in a bottom-up parser.
19. Define *left corner* and *trailing part.*
20. 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?
21. Summarize the tradeoffs between automatic and ad hoc management of space for attributes in a top-down parser.
22. At any given point in a top-down parse, which symbols will have attribute records in an automatically managed attribute stack?

4 Semantic Analysis

4.8 Exercises

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

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

4.27 Fill in the missing lines in Figure ◎4.19.

4.28 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* $\longrightarrow \epsilon$. Be sure to indicate where lhs and rhs point in the attribute stack. Also show the stack of saved lhs and rhs values, showing where each points in the attribute stack. You may ignore the ssf (seen so far) pointer.

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

$$
\begin{aligned}
&\textit{block} \longrightarrow \texttt{begin}\ \textit{stmt\_list}\ \texttt{end}\\
&\textit{stmt\_list} \longrightarrow \textit{stmt}\ \textit{stmt\_list\_tail}\\
&\textit{stmt\_list\_tail} \longrightarrow\ ;\ \textit{stmt\_list} \mid \epsilon\\
&\textit{stmt} \longrightarrow \ldots
\end{aligned}
$$

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 can you do if your program causes the compiler to run out of stack space? How can you modify your program to "get around" the problem?

4 Semantic Analysis

4.9 Explorations

4.34 As described in Section ©4.5.1, `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?

4.35 Learn how attribute space is managed in the ANTLR parser generator. How does it compare to the techniques described in Section ©4.5.2?

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 often 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 5.4: the development of microprogramming in the early 1960s, the development of the microprocessor in the early to mid-1970s, the development of RISC machines in the early 1980s, and the move to multithreaded and multicore processors in the first decade of the 21st century.

Most of the discussion in this chapter, and indeed in the rest of the book, will assume that we are compiling for a single-threaded RISC (reduced instruction set computer) architecture. Roughly speaking, a RISC machine is one that sacrifices richness in the instruction set in order to increase the number of instructions that can be executed per second. Where appropriate, we will devote a limited amount of attention to earlier, CISC (complex instruction set computer) architectures. The most popular desktop processor in the world—the x86—is a legacy CISC design, but RISC dominates among newer designs, and modern implementations of the x86 generally run fastest if compilers restrict themselves to a relatively simple subset of the instruction set. Within a modern x86 processor, a hardware "front end" translates these instructions, on the fly, into a RISC-like internal format.

In the first three sections below 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 to 5 of Patterson and Hennessy's outstanding text [PH08]).

	Typical access time	Typical capacity
Registers	0.2–0.5 ns	256–1024 bytes
Primary (L1) cache	0.4–1 ns	32 K–256 K bytes
Secondary (L2) cache	4–10 ns	1–8 M bytes
Tertiary (off-chip, L3) cache	10–50 ns	4–64 M bytes
Main memory	50–500 ns	256 M–16 G bytes
Disk	5–15 ms	80 G bytes and up
Tape	1–50 s	effectively unlimited

Figure 5.1 The memory hierarchy of a workstation-class computer. Access times and capacities are approximate, based on 2008 technology. Registers must be accessed within a single clock cycle. Primary cache typically responds in 1 to 2 cycles; off-chip cache in more like 20 cycles. Main memory on a supercomputer can be as fast as off-chip cache; on a workstation it is typically much slower. Disk and tape times are constrained by the movement of physical parts.

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

5.1 The Memory Hierarchy

EXAMPLE 5.1
Memory hierarchy stats

Memory on most machines consists of a numbered sequence of 8-bit bytes. It is not uncommon for modern workstations to contain several gigabytes of memory—much too much to fit on the same chip as the processor. Because memory is off-chip (typically on the other side of a bus), getting at it is much slower that getting at things on-chip. 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 ©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 RISC machine has two sets of registers, to hold integer and floating-point operands. It also has several special purpose registers, 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 ◎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 ◎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 typically 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 caches are larger and slower, but still faster than main memory. In a modern desktop or laptop system they are typically also on the same chip as the processor. High-end desktop or server-class machines may have an off-chip tertiary (L3) cache as well. Multicore processors, which have more than one processing core on a single chip, may share the L2 among cores, or even introduce an on-chip L3. Small embedded processors may have ony a single level of on-chip cache, with or without any off-chip cache. 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

DESIGN & IMPLEMENTATION

The processor/memory gap

Historically processor speed has increased much faster than memory speed, so the number of processor cycles required to access memory has continued to grow. As a result of this trend, caches have become 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.

or main memory. Cache lines vary from as few as 8 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, and 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. Most 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, which for reasons of backward compatibility allows operands to appear at arbitrary addresses, runs faster if those operands are properly aligned. Second, on RISC machines, there are generally not enough bits in an instruction to specify both an operation (e.g., `load`) and a full address. As we shall see in Section ©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.

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 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

EXAMPLE 5.2
Big- and little-endian

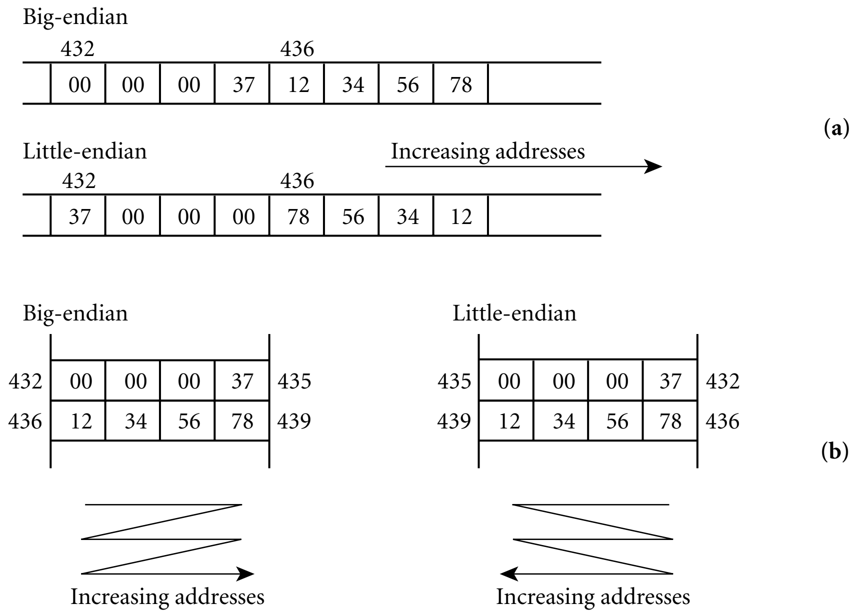


Figure 5.2 **Big-endian and little-endian byte orderings.** (a) Two 4-byte quantities, the numbers 37_{16} and $12\,34\,56\,78_{16}$, stored at addresses 432 and 436, respectively. (b) The same situation, with memory visualized as a byte-addressable array of words.

little-endian approach is that it seems to scramble the bytes of integers, when read from left to right (see Figure ◎5.2a). Little-endian-ness makes a bit more sense if one thinks of memory as a (byte-addressable) array of words (Figure ◎5.2b). Among CISC machines, the x86 is little-endian, as was the Digital VAX. The IBM 360/370 and the Motorola 680x0 are big-endian. Most first-generation RISC machines were also big-endian; most current RISC machines can run in either mode. ■

Support for characters varies widely. Most CISC machines will perform arbitrary arithmetic and logical operations on 1-byte quantities. Many CISC machines also provide instructions that perform operations on strings of characters, such as copying, comparing, or searching. Most RISC machines will load and store bytes from or to memory, but operate only on longer quantities in registers.

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).

An n-bit unsigned integer with binary representation $b_{n-1}\, b_{n-2}\, \ldots\, b_2\, b_1\, b_0$ has the value $\sum_{0 \leq i < n} b_i 2^i$. 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 ◎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 ◎5.3, the hexadecimal value `0xabcd` corresponds to the binary value `1010 1011 1100 1101`. Similarly, `0x400` $= 2^{10} = 1024$, commonly written `1K`, and `0x100000` $= 2^{20} = 1048576$, commonly written `1M`. ■

EXAMPLE 5.3
Hexadecimal numbers

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 modulo 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,

DESIGN & IMPLEMENTATION

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 personal 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 hard disk will hold 10^{11} bytes.

0000	0	1000	8
0001	1	1001	9
0010	2	1010	a
0011	3	1011	b
0100	4	1100	c
0101	5	1101	d
0110	6	1110	e
0111	7	1111	f

Figure 5.3 The hexadecimal digits.

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.

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 ©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. The only difference is the mechanism used to detect whether *overflow* has occurred. 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 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.

If we ignore 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 ignore). 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.[1] ■

EXAMPLE 5.5
Overflow in two's complement addition

Different machines handle overflow in different ways. Some generate a trap (an interrupt) 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 ©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 is 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 $k + (-k)$.

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 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, underflow, overflow, invalid operations, and the representation of extremely small quantities. With the completion in 1985 of IEEE standard number 754, however, the situation changed dramatically. Most processors developed in subsequent years conform to the formats and semantics of this standard.

1 Exercise ©5.6 considers an alternative but equivalent definition, which is particularly easy to test in hardware.

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

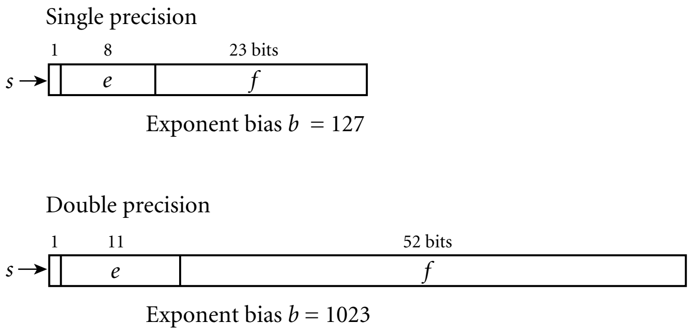
Most values in the IEEE standard are *normalized* by shifting the significant until it is greater than or equal to 1, and less than 2. (The exponent is adjusted accordingly, so that the value represented doesn't change.) As a result of normalization, the number of bits in the significand is really one more than the number explicitly represented: in the value 1. *something* $\times\ 2^{exp}$, the one is superfluous, and is omitted in the representation. Exceptions to this rule occur near zero: very small numbers can be represented (with reduced precision) as 0.*something* $\times\ 2^{min+1}$, where *min* is the smallest (most negative) exponent available in the format. Many older floating-point standards disallow such *denormal 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 denormals, the IEEE standard is said to provide for *gradual underflow*. Denormal numbers are represented with a zero in the exponent field (denoting a maximally negative exponent) together with a nonzero fraction.

EXAMPLE 5.7
IEEE floating-point

The conventions of the IEEE 754 standard are summarized in Figure 5.5. In addition to single- and double-precision formats, the standard also provides for vendor-defined "extended" single- and double-precision numbers (not shown here). These extended formats are required to have at least 32 and 64 significant bits (31 and 63 explicit) in the significand, respectively. ■

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:

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



	e	f	Value
Zero	0	0	± 0
Infinity	$2b+1$	0	$\pm \infty$
Normalized	$1 \le e \le 2b$	<*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 denormal (gradual underflow) numbers.

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 an interrupt if used.

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 [HP07, App. I].

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 double-precision values?
6. What is a floating-point NaN?

5.3 Instruction Set Architecture

On a RISC machine, computational instructions operate on values held in registers: 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 RISC machines generally specify three registers: two sources and a destination. Some CISC machines (e.g., the VAX) also provide three-address instructions. Others (e.g., the x86 and the 680x0) 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 allow both operands to be reused in subsequent operations. This reuse is crucial on RISC machines: it minimizes the number of artificial restrictions on the ordering of instructions, affording the compiler considerably more freedom in choosing an order that performs well.

5.3.1 Addressing Modes

One can imagine many different ways in which a computational instruction might specify the location of its operands. A given operand might be in a register, in memory, or, in the case of read-only constants, in the instruction itself. If the 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.

As noted above, most RISC machines require that the operands of computational instructions reside in registers or the instruction. For load and store instructions, which are allowed to access memory, they typically support 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 RISC machines, including the PowerPC and SPARC, 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 a richer 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 are generally controlled in one of two ways. On most CISC machines they use *condition codes.* As mentioned in Section 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 assembler

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

a compiler for the x86[2] 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:
```

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:

2 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 assembler. 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 assembler the *destination* operand is listed second. In Intel and Microsoft assembler 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`.

```
                        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, but it does *not* 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 some important performance enhancements. In particular, the fact that they are set by almost every instruction tends to preclude implementations in which logically unrelated instructions might be executed in between (or in parallel with) the instruction that tests a condition and the branch that relies on the outcome of the test. There are several possible ways to address this problem; the handling of conditional branches is one of the areas in which extant RISC machines vary most from one another. The ARM and SPARC architectures make setting of the condition codes optional on an instruction-by-instruction basis. The PowerPC provides eight separate sets of condition codes; compare and branch instructions can specify the set to use. The MIPS has no condition codes (at least not for integer operations); it uses Boolean values in registers instead.

EXAMPLE 5.10

Conditional branches on the MIPS

More precisely, where the x86 has 16 different branch instructions based on arithmetic comparisons, the MIPS has only six. Four of these branch if the value in a register is $<$, $\leq$, $>$, or $\geq$ zero. The other two branch if the values in two registers are $=$ or $\neq$. In a convention shared by most RISC machines, register zero is defined to always contain the value zero, so the latter two instructions cover both the remaining comparisons to zero and direct comparisons of registers for equality. More general register-register comparisons (signed and unsigned) require a separate instruction to place a Boolean value in a register that is then named by the branch instruction. Repeating the examples above on the MIPS, we get

```
                        lw      $3, A        ; load word: register 3 := A
if A ≤ B then           lw      $2, B        ; register 2 := B
    body                slt     $2, $2, $3   ; register 2 := (B < A)
                        bne     $2, $0, L1   ; branch if Boolean true (≠ 0)
                        body
                    L1:
```

```
if A > 0 then              blez    $3, L2      ; branch if A ≤ 0
    body                   body
                       L2:
```

By convention, destination registers are listed first in MIPS assembler (as they are in assignment statements). The `slt` instruction stands for "set less than"; `bne` and `blez` stand for "branch if not equal" and "branch if less than or equal to zero," respectively. Note that the compiler has used `bne` to compare register 2 to the constant register 0. ■

✓ CHECK YOUR UNDERSTANDING

7. What is the world's most popular instruction set architecture (for desktop machines)?
8. What is the difference between big-endian and little-endian addressing?
9. What is the purpose of a cache?
10. Why do many machines have more than one *level* of cache?
11. How many processor cycles does it typically take to access primary (on-chip) cache? How many cycles does it typically take to access main memory?
12. What is data *alignment*? Why do many processors insist upon it?
13. List four common formats (interpretations) for bits in memory.
14. What is IEEE standard number 754? Why is it important?
15. What are the tradeoffs between two-address and three-address instruction formats?
16. Describe at least five different addressing modes. Which of these are commonly supported on RISC machines?
17. 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. Using ideas originally developed for supercomputers of the 1960s, mainstream processors of the 1980s and 1990s increasingly moved toward a *pipelined* organization, in which the functional units work like the stations on an assembly line, with different instructions passing through different pipeline stages concurrently. Pipelining is used in even the most inexpensive personal computers today, and in all but the simplest processors for the embedded market. A simple processor may have five or six pipeline stages. The IBM PowerPC G5 has 21; the Intel Pentium 4E has 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 ◎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. An important example occurs in the case of floating-point arithmetic, which is typically 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. A major problem with the trend toward more complex instruction sets was that it made it difficult to design high-performance implementations. On the VAX, for example (the most popular minicomputer of the early 1980s), instructions could vary in length from one to more than 50 bytes, and in execution time from one to thousands of cycles. Both of these factors tend to lead to pipeline stalls. Variable length instructions make it difficult to even find the next instruction until the current one has been studied extensively. Variable execution time makes it difficult to keep all the pipeline stages busy. The original VAX (the 11/780) was shipped in 1978, but it wasn't until 1985 that Digital was able to ship a successfully pipelined version, the 8600.[3]

The most basic rule of processor performance holds that total execution time on any machine equals the number of instructions executed times the average number of cycles per instruction times the length in time of a cycle. 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. The "RISC philosophy," by contrast, is to minimize execution time by reducing the length of the cycle and the number of (nonoverlapped) cycles per instruction (CPI).

Recent RISC machines (and RISC-like implementations of the x86) attempt to minimize CPI by executing as many instructions as possible in parallel. The PowerPC G5, for example, can have over 200 instructions simultaneously "in

3 An alternative approach, to maintain microprogramming but pipeline the microengine, was adopted by the 8800 and, later, by Intel's Pentium Pro and its successors.

flight." 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. High performance processor implementations may actually execute mutually independent instructions *out of order* when they 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 40 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 over that period of time. 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 of 2 GHz (1000× faster) are common 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 the sidebar on page ◎67, 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 130 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, we're executing as many instructions in parallel as our programs will allow.

Robbed of the ability to run a single program faster, processor designers have taken to building *multithreaded* and *multicore* chips that can run more than one program at once. Historically, 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 in the processor's pipeline. A multicore processor, by contrast has two or more complete processors (cores) on a single chip. Compared to a high-end uniprocessor, these may run at a somewhat slower clock rate, and expend less energy on speculation and ILP discovery, in order to maximize performance per watt.

As of the summer of 2008, Intel sells processors with two dual-core chips in a single package, and is rumored to be working on a six-core design for release in the next few months. AMD and IBM are both selling quad-core processors today. Sun, which specializes in servers that have naturally parallel workloads, is currently leading the pack, with 8-core, 64-thread chips available today, and 16-core chips expected by the end of the year.

In moving to multicore processors, the computer industry has effectively given up on running conventional programs faster, and is banking instead on running better programs. This makes the current revolution in processor design very different from its predecessors. Where previous revolutions were mostly invisible to programmers (code might perhaps have to be recompiled to make the best use of a new machine), the current revolution will eventually require that programs be *rewritten* in some concurrent programming language. And while successes in high-end scientific and commercial computing have demonstrated that this task is possible for expert programmers in certain problem domains, it is not yet clear whether it will be possible for "ordinary" programmers in multiple problem domains.

Most computers do several things at once: they update the display, check for mail, play music, and execute user commands by switching the processor from one task to another many times a second. With several cores available, each task can run on a different core, reducing the need for switching. But what will happen when we have 100 cores? Where will we find 100 runnable threads? This is perhaps the most vexing problem currently facing the field of computer systems.

✓ **CHECK YOUR UNDERSTANDING**

18. What is microprogramming? What breakthroughs did its invention make possible?
19. 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?
20. What is pipelining?
21. Summarize the difference between the CISC and RISC philosophies in instruction set design.
22. Why do RISC machines allow only load and store instructions to access memory?
23. Name three CISC architectures. Name three RISC architectures. (If you're stumped, see the Summary and Concluding Remarks [Section ◎5.6].)
24. What three research groups share the credit for inventing RISC? (For this you'll probably need to peek at the Bibliographic Notes [Section ◎5.9].)
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. Explain why the multicore revolution poses an unprecedented challenge for computer systems.

5.4.5 Two Example Architectures: The x86 and MIPS

EXAMPLE 5.11
The x86 ISA

We can illustrate the differences between CISC and RISC machines by examining a representative pair of architectures. The x86 is the most widely used CISC design—in fact, the most widely used processor architecture of any kind (outside the embedded market). The original model, the 8086, was announced in 1978. Major changes were introduced by the 8087, 80286, 80386, Pentium Pro, Pentium/MMX, Pentium III, and Pentium 4. While technically backward compatible, these changes were often out of keeping with the philosophy of the earlier generations. The result is a machine with an enormous number of stylistic inconsistencies and special cases. The 64-bit extension to the x86 was likewise saddled with the need for backward compatibility, and is even more complex. It was originally developed by AMD and subsequently licensed by Intel. (AMD calls it AMD64. Intel calls it Intel 64. Generically, it is often referred to as x86-64, or simply x64).

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 MIPS ISA

The MIPS architecture, begun as a commercial spin-off of research at Stanford University, is arguably the simplest of the commercial RISC machines. It too has evolved, through six principal generations (the last in 1999). With one exception, however—the introduction of 64-bit integer operands and addresses in 1991—the changes have been relatively minor. Current processors are sold in 32- and 64-bit versions. MIPS processors were used by several workstation vendors—notably Silicon Graphics—throughout the 1990s. They are now used primarily in embedded applications. MIPS-based tools are also widely used in academia. All models of the MIPS are implemented entirely in hardware; they are not microprogrammed. ■

Among the most significant differences between the x86 and MIPS are their memory access mechanisms, their register sets, and the variety of instructions they provide. Like all RISC machines, the MIPS allows only load and store instructions to access memory; all computation is done with values in registers. Like most CISC machines, the x86 allows computational instructions to operate on values in either registers or memory. It also provides a richer set of addressing modes. Like most RISC machines, the MIPS has 32 integer registers and 32 floating-point registers. The 32-bit x86, by contrast, has only 8 of each, and most of the floating-point instructions treat the floating-point registers as a tiny stack, rather than naming them directly. The MIPS 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. All MIPS instructions are exactly 4 bytes long. Instructions on the x86 vary from 1 to 17 bytes.

Memory Access and Addressing Modes

Like all RISC machines, the MIPS has a load/store architecture. Its memory access instructions support only displacement addressing. With a displacement of zero this subsumes register indirect addressing. With a base of zero (hardwired into register zero), it also subsumes so-called *absolute* addressing in the first 64K of memory. On the x86, by contrast, most instructions can obtain one (but not both) of their operands from memory. Nine different addressing modes are available for these references. The most general case is called *scaled indexed* addressing. It employs a base register R_b, an index register R_i, a displacement d, and a scaling factor s. The value of s must be 1, 2, 4, or 8; it can therefore be encoded in two bits. The effective address of the operand is $(R_b) + d + (R_i) \times s$, where (R) represents the content of register R. All of the other x86 addressing modes are simplifications of this general case.

MIPS instructions are three-address. X86 instructions are two-address: the result of a computation overwrites one of the operands, which may be in either a register or memory.

Registers

EXAMPLE 5.13
x86 and MIPS register sets

The register sets of the two machines are illustrated pictorially in Figure 5.6. The most striking difference between the two is the sheer number of registers on the RISC machine—roughly four times as many as on the CISC machine. The integer registers are also wider on the MIPS, but both machines have "widened" over time. 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. The MIPS, by contrast, was introduced with 32-bit integer registers in 1984. These were expanded to 64 bits in 1991. The x86-64 architecture, introduced by AMD in 2003, extends the x86 to 64-bit registers. We do not consider those extensions here; even on 64-bit machines, most programs continue to run in 32-bit mode. ■

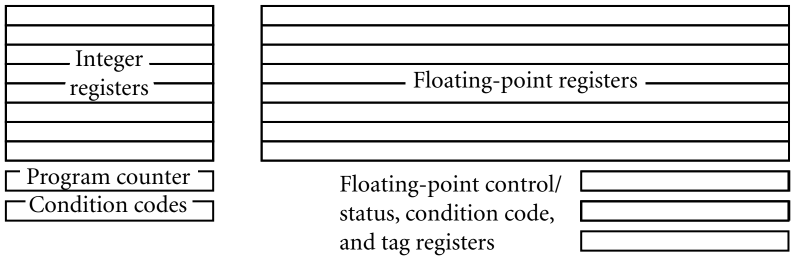
The x86 has eight 32-bit integer registers, plus the program counter and the processor status word, which includes the condition codes. For historical reasons, the integer registers are named `eax`, `ebx`, `ecx`, `edx`, `esi`, `edi`, `esp`, and `ebp`. They can be used interchangeably in most instructions, 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`.

Floating-point instructions manipulate a separate set of 80-bit floating-point registers. There are also registers for 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, denormal, NaN, or garbage. All computation is carried out in extended precision; values are converted to and from IEEE single- and double-precision floating-point when written to or read from memory.

Recent members of the x86 processor family support instruction set extensions (MMX and SSE) that allow arithmetic operations to be performed on vectors of small integer or floating-point operands. While we will not consider these extensions further, it is worth mentioning that the eight MMX vector registers overlap the low-order 64 bits of the x86 floating-point registers. The eight SSE vector registers are new; each is 128 bits long.

The MIPS has a total of 64 registers, 32 integer and 32 floating-point, plus the program counter; a pair of special registers, called `LO` and `HI`, used by multiply and divide instructions; a floating-point control and status register analogous to the



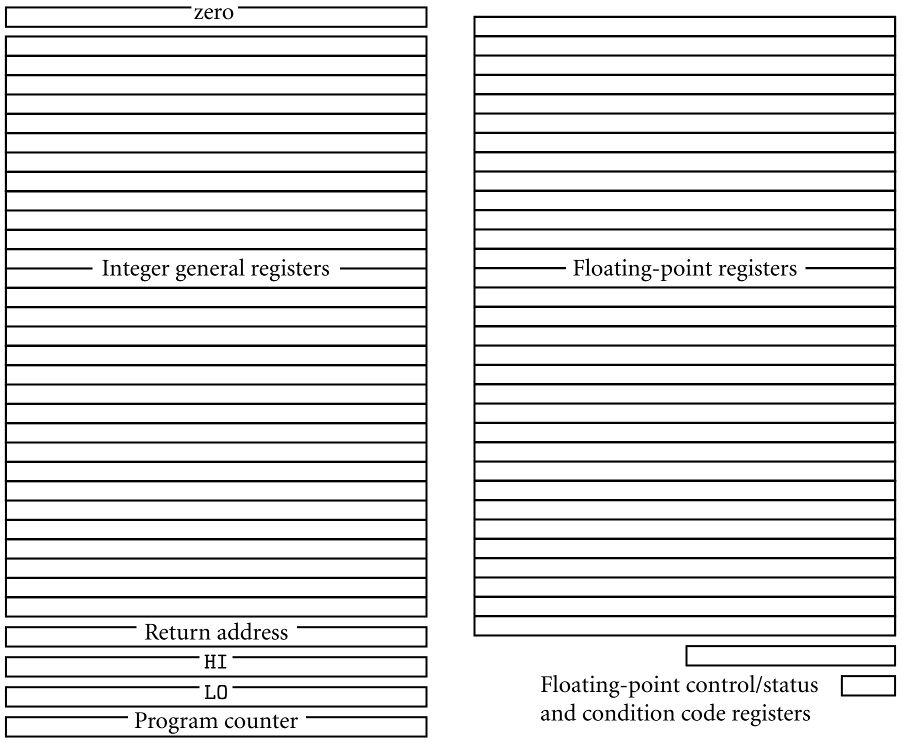

zero
Integer general registers
Floating-point registers
Return address
HI
LO
Program counter
Floating-point control/status and condition code registers

Figure 5.6 Register sets of the x86 (top) and MIPS IV (bottom). In both cases, only those registers of interest to the user-level programmer are shown; implementations of both architectures include special-purpose registers of use to the operating system. Not shown are eight 128-bit "streaming registers" introduced with the SSE extensions to the x86 and a 192-bit accumulator introduced with the MDMX extensions to the MIPS. Also omitted are eight segment registers in the x86 that support the obsolete 80286 addressing system; these are not used by modern compilers.

one on the x86; and an eight-bit floating-point condition code register. Branches based on the outcome of integer operations employ combination instructions that test or compare values in registers and branch based on the outcome; there are no integer condition codes. In early generations of the MIPS, integer registers were 32 bits wide; in more recent generations they are 64. Double-precision (64-bit) floating-point arithmetic has always been available, but in early generations it required that the floating-point registers be used together in pairs.

Integer register `r0` on the MIPS always contains a zero. This design trick allows several simplifications in instruction encoding. To move a value from one register to another, for example, we can perform an `add` (or a `sub` or an `or`) with `r0`; we don't need a separate instruction. To negate a value we can subtract it from `r0`. To branch unconditionally, we can "test" whether `r0` = `r0`. The only other nonuniformity in the treatment of MIPS registers appears in the subroutine call instruction, `jal` (jump and link): it places its return address in register `r31`.

Integer multiply instructions on the MIPS write their results to registers `LO` and `HI` (with n-bit operands, the result of a multiply may require $2n$ bits). Divides always generate both a quotient and a remainder, in `LO` and `HI` respectively. Special `move` instructions copy from `LO` and/or `HI` to/from integer registers. As noted above, the x86 overloads registers `eax` and `edx` for these purposes.

In a manner analogous to the MMX and SSE extensions to the x86, recent MIPS processors also support small integer and floating-point vector operations. Data for these operations are kept in the floating-point registers, and in a new 192-bit vector accumulator that allows the results of a series of integer vector multiplies to be totaled without overflow.

Register Conventions Beyond the special treatment given some registers in hardware, the designers of both the x86 and the MIPS recommend additional conventions to be enforced by software. On the x86, 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. (Calling sequences will be discussed in more detail in Section 5.5.2.)

Conventions are even more elaborate on the MIPS. Register `r1` is reserved for the assembler, which uses it when expanding certain *pseudoinstructions* into sequences of real instructions. Registers `r2` and `r3` are used for expression evaluation and function returns. Registers `r4..r7` are used for subroutine parameters. Registers `r16..r23` are "callee saves" registers comparable to `ebx`, `esi`, and `edi` on the x86. Registers `r8..r15`, `r24`, and `r25` are "caller saves" registers. Registers `r26` and `r27` are reserved for use by the operating system kernel; from the point of view of a user program their values can change spontaneously. Register `r28` is used as a base for displacement addressing of global variables. Registers `r29` and `r30` are used for the stack pointer and frame pointer, respectively.

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 the MIPS. Some of the features of the x86 not found in the MIPS include:

- Binary-coded decimal arithmetic (see the sidebar on page 296).
- Character-string search, compare, and copy operations.
- Bit test and set 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 byte and register swaps, subroutine calls and returns, stack operations, and loop control.
- Instructions to support the obsolete 80286 segmented memory system.

On the other hand, the MIPS provides:

- "Building-block" instructions that allow a 32-bit quantity to be loaded into a register with a two-instruction sequence.
- Separate 32- and 64-bit versions of most of the arithmetic, logical, and memory-access (load/store) instructions.
- Nullifying branches (discussed in Section ©5.5.1).
- Conditional traps. These provide a fast way to drop into the operating system on a dynamic semantic error.

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 17 bytes in length, with a myriad 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. 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.

Floating-point operations are perhaps the most baroque component of the x86 instruction set. The designers of the original floating-point co-processor, the Intel 8087, chose to conserve space in floating-point instructions by treating the floating-point registers as a stack. A floating-point load instruction, for example, does not specify a destination register; instead it pushes its operand into the top location on the register stack, designated `st(0)`. The previous contents of registers `st(0)` through `st(6)` move down so that what used to be in `st(4)`, for example, is now in `st(5)`. The content of `st(7)` is lost. Arithmetic instructions, likewise, manipulate the stack. The simplest `add` instruction, for example, does not specify operand or destination registers; instead it reads its operands implicitly from `st(0)` and `st(1)`, and replaces the content of `st(0)` with the result. A variant of the `add` instruction pops its operands: the contents of registers `st(2)` through `st(7)` move up one position, and `st(0)` gets the result of adding the old `st(0)` and `st(1)`. Additional instruction variants take one operand from an explicitly named register farther down the stack, or from a location in memory. Noncommutative operations, such as subtraction and division, have variants that use their operands in either order.

While floating-point arithmetic instructions (and compare and test instructions) update the floating-point condition codes, there are no special branch instructions to alter control flow based on these codes. Instead, an `fnstsw` instruction must be used to copy the floating-point status word to one of the 16-bit integer registers, where a subsequent bit test instruction can access the desired condition codes. This instruction in turn sets the integer condition codes, which can be used to direct a branch.

Like most RISC machines, the MIPS employs fixed-length, 32-bit instructions with relatively simple encodings. The first six bits specify the opcode. The remaining bits contain (1) a 26-bit jump displacement, (2) a pair of register specifiers and a 16-bit constant, or (3) three register specifiers and 11 additional bits, some of which are used by special-purpose instructions. Many of the register-register operations contain unused bits, and operations that might be specified with, say, 40 bits on the x86 require two full instructions (64 bits) on the MIPS. Like the x86, the MIPS implements the IEEE 754 floating-point standard, but its floating-point instruction set is much more simple and straightforward—very similar to the integer set. Floating-point branch instructions have direct access to the floating-point condition codes.

✓ CHECK YOUR UNDERSTANDING

29. Describe the most general (complex) addressing modes of the x86 and MIPS architectures.

30. How many integer and floating-point registers are provided by each machine? How wide are these registers?

31. Explain the utility of register zero on the MIPS.

32. Summarize the register usage conventions of the x86 and MIPS.

33. List at least three "complex" instructions provided by the x86 instruction set but not provided by the MIPS instruction set.

34. Name a "simple" instruction provided by the MIPS but not by the x86.

35. Describe the floating-point stack of the x86.
36. Summarize the differences in instruction encoding between the x86 and MIPS.

5.5 Compiling for Modern Processors

Programming a RISC machine by hand, in assembly language, is a tedious undertaking. Only loads and stores can access memory, and then only with limited addressing modes. Moreover the limited space available in fixed-size instructions means that a nonintuitive two-instruction sequence is required to load a 32-bit constant or to jump to an absolute address. In some sense, complexity that used to be hidden in the microcode of CISC machines 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.

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. A user who traded in 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. ■

EXAMPLE 5.14
Performance $\neq$ clock rate

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 are 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.[4] 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.

Of course, even out-of-order execution works only if the processor is able to fetch instructions, and thus it is control hazards that have the largest potential negative impact on performance. Branches constitute something like 10% of all instructions in typical programs,[5] 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 workstation-class processors incorporate hardware to *predict* the outcome of each branch, based on past behavior, and to execute speculatively down the predicted path. Assuming that it takes care to avoid any irreversible operations, the processor will suffer stalls only in the case of an incorrect prediction.

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 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

4 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.

5 This is a very rough number. For the SPEC2000 benchmarks, Hennessy and Patterson report percentages varying from 1 to 25 [HP07, 3rd ed., pp. 138–139].

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 processor 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.

Instruction scheduling can have a major impact on resource and data hazards. On machines with so-called *delayed branches* it can also help with control hazards. We will consider the topic of instruction scheduling in some detail in Section ©16.6. In the remainder of the current subsection we focus on the two cases—loads and branches—where issues of instruction scheduling may actually be embedded in the processor's instruction set. Software techniques to reduce the incidence of cache misses typically require large-scale restructuring of control flow or data layout. Though the better commercial compilers may reorganize loops for better cache locality in scientific programs (a topic we will consider in Section ©16.7.2), most simply assume that every memory access will hit in the primary cache. The assumption is generally a good one: most programs on most machines achieve a cache hit rate of well over 90% (often over 99%). The important goal is to make sure that the pipeline can continue to operate during the time that it takes the cache to respond.

Loads

Consider a load instruction that hits in the primary cache. The number of cycles that must elapse before a subsequent instruction can use the result is known as the *load delay*. Most current machines have 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.

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 will incur a one-cycle penalty between the second and third instructions:

EXAMPLE 5.15
Filling a load delay slot

```
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 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).

Branches

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 for straight-line code, but not for the code that follows a branch.[6] 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 is executed regardless of the outcome of the branch. If the branch is not taken, all occurs as one would normally expect. If the branch is taken, however, the order of instructions is the branch itself, the instruction after the branch, and then the instruction at the target of the branch.

EXAMPLE 5.17
Filling a branch delay slot

Because control may go either of two directions at a branch, finding an instruction to fill a delayed branch slot is slightly trickier than finding one to fill a delayed load slot. The few instructions immediately before the branch are the most obvious candidates to move, provided that they do not contribute to the calculation that controls the branch, and that we don't have to move them past the target of some other branch:

```
B := r2                 -- movable ↓
r1 := r2 * r3           -- immovable ↓̸                r1 := r2 * r3
if r1 > 0 goto L1                          becomes     if r1 > 0 goto L1
nop                                                    B := r2
```

(This code sequence assumes that branches are delayed. Unless otherwise noted, we will assume throughout the remainder of the book that they are not.) ■

To address the problem of unfillable branch delay slots, some RISC machines provide *nullifying* conditional branch instructions. A nullifying branch includes a bit that indicates the direction that the compiler "expects" the branch to go. The hardware executes the instruction in the delay slot only if the branch goes

DESIGN & IMPLEMENTATION

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 elsewhere in this section. 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`)—an instruction that has no effect. 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.

6 In this context, branches include not only the control flow for conditionals and loops, but also subroutine calls and returns.

the expected direction. While the branch instruction is making its way down the pipeline, the hardware begins to execute the next instruction. Ideally, by the time it must begin the instruction after that, it will know the outcome of the branch. If the outcome matches the prediction, then the pipeline will proceed without stalling. If the outcome does not match the prediction, then the (not yet completed) instruction in the delay slot will be abandoned, along with any instructions fetched from the target of the branch.

Unfortunately, as architects have moved to more aggressive, deeply pipelined processor implementations, multicycle branch delays have become the norm, and architecturally visible delay slots no longer suffice to hide them. A few processors have been designed with an architecturally visible branch delay of more than one cycle, but this is not generally considered a viable strategy: it is simply too difficult for the compiler to find enough instructions to schedule into the slots. Several processors retain one-slot delayed branches (sometimes with optional nullification) for the sake of backward compatibility, and as a means of reducing, but not eliminating, the number of pipeline stalls (the *penalty*) associated with a branch. With or without delayed branches, many processors also employ elaborate hardware mechanisms to predict the outcome and targets of branches early, so that the pipeline can continue anyway. When a prediction turns out to be incorrect, of course, the hardware must ensure that none of the incorrectly fetched instructions have visible effects. 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 in the most likely cross-branch code paths.

5.5.2 Register Allocation

The load-store architecture of RISC machines 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. These same costs are present on CISC machines as well, even if they don't stand out as prominently in a casual perusal of assembly code. 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 (page 5) and the naive code of Figure 1.6 (page 34) is the absence in the former of most of the loads and stores. As improvements in processor speed continue to outstrip improvements in memory speed, the cost in cycles of a cache miss continues to increase, making good register usage increasingly important.

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 of the machine, 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.

We will examine this two-stage process in more detail in Section ©16.8. 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,

EXAMPLE 5.18
Register allocation for a simple loop

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

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 RISC machine is likely to look something like Figure ©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 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 of the machine. 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

```
         v1 := &A            -- pointer to A[1]
         v2 := n             -- count of elements yet to go
         w1 := 0.0           -- sum
         w2 := 0.0           -- squares
         goto L2
    L1:  w3 := *v1           -- A[i] (floating point)
         w1 := w1 + w3       -- accumulate sum
         w4 := w3 × w3
         w2 := w2 + w4       -- accumulate squares
         v1 := v1 + 8        -- 8 bytes per double-word
         v2 := v2 – 1        -- decrement count
    L2:  if v2 > 0 goto L1
         w5 := w1 / n        -- average
         w6 := w2 / n        -- average of squares
         w7 := w5 × w5       -- square of average
         w8 := w6 – w7
         . . .               -- return value in w8
```

Figure 5.7 **RISC assembly code for a vector variance computation.**

```
         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.

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 5.8. ■

```
        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.

Interaction with Instruction Scheduling

From the point of view of execution speed, the code in Figure 5.8 has at least two problems. First, of the seven instructions in the loop, nearly half are devoted to bookkeeping: updating the pointer, decrementing the loop count, and testing the terminating condition. Second, when run on a pipelined machine, the code is likely to experience a very high number of stalls. Exercise 5.23 explores a first step toward addressing the bookkeeping overhead. We consider the stalls below, and return to both problems in more detail in Chapter 16.

EXAMPLE 5.19
Register allocation and instruction scheduling

We noted in Section 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 the usual 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 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. ■

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 will 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.

DESIGN & IMPLEMENTATION

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.

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, the compiler 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* in more detail in Chapter 8.

✓ CHECK YOUR UNDERSTANDING

37. What is a *delayed load* instruction?
38. What is a *nullifying branch* instruction?
39. List the four principal causes of pipeline stalls.
40. What is a pipeline *interlock*?
41. What is *instruction scheduling*? Why is it important on modern machines?
42. What is *branch prediction*? Why is it important?
43. Describe the interaction between instruction scheduling and register allocation.
44. What is the *live range* of a register?
45. What is *subroutine inlining*? What benefits does it provide? When is it possible? What is its cost?
46. 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 acceptable 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 RISC machines. As examples we compared the x86 and the MIPS. We also introduced a simple notation to be used for assembly language examples in later chapters. In the final section we discussed why compiling for modern machines is hard. The principal tasks include *instruction scheduling*, for load and branch delays and for 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.

As of 2009 there are four principal commercial RISC architectures: ARM (Marvell, Texas Instruments, Motorola, and dozens of others), MIPS (NEC, Toshiba, Freescale, and many others), Power/PowerPC (IBM, Freescale), and SPARC (Sun, Texas Instruments, Fujitsu, and several others). ARM is the property of ARM Holdings, PLC, an intellectual property firm that relies on licensees for actual fabrication. Though ARM processors are not generally employed in desktop or laptop computers, they power roughly three-quarters of the world's embedded systems, in everything from cell phones and PDAs to remote controls and the dozens of devices in a modern automobile. MIPS processors, likewise, are now principally employed in the embedded market, though they were once common in desktop and high-end machines.

Despite the handicap of a CISC instruction set and the need for backward compatibility, the x86 overwhelmingly dominates the desktop and laptop market, largely due to the marketing prowess of IBM, Intel, and Microsoft, and to the success of Intel and AMD in decoupling the architecture from the implementation. Modern implementations of the x86 incorporate a hardware front end that translates x86 code, on the fly, into a RISC-like internal format amenable to heavily pipelined execution. Recent processors from Intel and AMD are competitive with the fastest RISC alternatives.

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 Pentium processors.

As processor and compiler technology continue to evolve, it is likely that processor implementations will continue to become more complex, and that compilers will take on additional tasks in order to harness that complexity. Traditional CISC machines remain popular almost entirely due to the need for backward compatibility, but both the CISC and RISC "design philosophies" are still very much alive [SW94]. The "CISC-ish" philosophy says that newly available resources (e.g., increases in chip area) should be used to implement functions that must currently be performed in software, such as vector or graphics operations, decimal arithmetic, new addressing modes, or perhaps transactional memory (to be described in Section 12.4.4). The "RISC-ish" philosophy says that resources should 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 the cycle time.

Where the first-generation RISC machines from different vendors differed from one another only in minor details, later generations diverged, with the ARM and MIPS taking the more RISC-ish approach, the Power/PowerPC family taking the more CISC-ish approach, and the SPARC somewhere in the middle. It is not yet clear which approach will ultimately prove most effective, nor is it even clear that this is the interesting question anymore. Heat dissipation and limited ILP are increasingly the main constraints on uniprocessor performance. In response to these constrains, most vendors are now pursuing multicore versions of their respective architectures. It is entirely possible that future processors will be highly heterogeneous, with multiple implementation strategies—or 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 25 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 such as a Pentium? What will it print on a big-endian machine such as a Sun?

```
unsigned short n = 0x1234;
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 ©5.2.1 we observed that overflow occurs in two's complement addition when we add two nonnegative numbers and obtain an apparently negative result, or add two negative numbers and obtain an apparently nonnegative 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 ©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 (e.g., the SPARC) 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? Why not variable 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 $0\,..\,2^n$ for unsigned arithmetic, $-2^{n-1}\,..\,2^{n-1}-1$ for signed arithmetic). Suppose we wish to branch on condition `A op B`, where `A` and `B` are unsigned binary numbers, for `op` $\in \{<, \leq, =, \neq, >, \geq\}$. Suppose we subtract `B` from `A`, using two's complement arithmetic. For each of the six conditions, indicate the logical combination of condition-code bits that should be used to trigger the branch. Repeat the exercise on the assumption that `A` and `B` are signed, two's complement numbers.

5.14 We implied in Section ©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 on-chip (primary) cache, with a penalty of 40 cycles to reach main memory. What is the contribution of cache misses to the average number of cycles per instruction? You may assume that instruction fetches always hit in the cache. Now suppose that we add an off-chip (secondary) cache that can satisfy 90% of the misses from the primary cache, at a penalty of only 10 cycles. What is the effect on cycles per instruction?

5.16 Many recent processors provide a *conditional move* instruction that copies one register into another if and only if the value in a third register is (or is not) equal to zero. Give an example in which the use of conditional moves leads to a shorter program.

5.17 The x86-64 architecture is backward compatible with the x86 instruction set, just as the x86 is backward compatible with the 16-bit 8086 instruction set. Less transparently, the IA-64 Itanium is capable of running legacy x86 applications in "compatibility mode." But recent members of the ARM and MIPS processor families support *new* 16-bit instructions as an *extension* to the architecture. Why might designers have chosen to introduce these new, less powerful modes of execution?

5.18 Consider the following code fragment in pseudo-assembler notation:

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

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

(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?

5.19 With the development of deeper, more complex pipelines, delayed loads and branches have become significantly less appealing as features of a RISC instruction set. Why is it that designers have been able to eliminate delayed loads in more recent machines, but have had to retain delayed branches?

5.20 Some processors, including the PowerPC and recent 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?

5.21 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?

5.22 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?

5.23 Consider the code in Figure ©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

5.24 Skip ahead to the sidebar on decimal types on page 296 of 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?

5.25 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.

5.26 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?

5.27 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 MIPS instruction sets. Discuss, from a compiler writer's point of view, the challenges and opportunities presented by the IA-64.

5.28 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.

5.29 The x86-64 architecture is a backward-compatible 64-bit extension of the x86. Find a manual or tutorial and learn about the instruction set. Describe the extensions it provides. Explain how it can execute 32-bit x86 instructions without an explicit "compatibility mode."

5.30 Several computers have provided more general versions of the *conditional move* instructions described in Exercise ◎5.16. Examples include the c. 1965 IBM ACS, the Cray 1, the HP PA-RISC, the ARM, and the Intel IA-64 (Itanium). General-purpose conditional execution is sometimes known as *predication*.

Learn how predication works in ARM or IA-64. Explain how it can sometimes improve performance even when it causes the processor to execute *more* instructions.

5.31 If you have access to computers of more than one type, 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?

5.32 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?

5.33 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.)

5.34 Translate the high-level pseudocode for vector variance (Example ◎5.18) 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.

5.35 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?

5.36 If you have access to a compiler that generates code for a machine with architecturally visible load delays, run some programs through it and evaluate the degree of success it has in filling delay slots (an unfilled slot will contain a `nop` instruction). What percentage of slots is filled? Suppose the machine had interlocked loads. How much space could be saved in typical executable programs if the `nop`s were eliminated?

5.37 Experiment with small subroutines in C++ to see how much time can be saved by expanding them inline.

5.9 Bibliographic Notes

The standard reference in computer architecture is the graduate-level text by Patterson and Hennessy [HP07]. More introductory material can be found in the undergraduate computer organization text by the same authors [PH08]. Students without previous assembly language experience may be particularly interested in the text of Bryant and O'Hallaron [BO03], 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 assembler.

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 PowerPC and Alpha architectures. Hennessy and Patterson's architecture text includes an appendix that summarizes the similarities and differences among the major commercial instruction sets [HP07, App. J]. 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 [Gol07]. 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 [CT04] are excellent sources of information on instruction scheduling, register allocation, subroutine optimization, and other aspects of compiling for modern machines.

6 Control Flow

6.5.4 Generators in Icon

EXAMPLE 6.85
Simple generator in Icon

Like the iterators of Clu, Python, Ruby, and C#, Icon generators can be used for enumeration-controlled iteration. Our canonical `for` loop example would be written as follows in Icon:

```
every i := first to last by step do {
    ...
}
```

Here ... `to`... `by`... is a built-in "mixfix" generator. ■

Because Icon is intended largely for string manipulation, most of its built-in generators operate on strings. `Find(substr, str)`, for example, generates the positions (indices) within string `str` at which an occurrence of the substring `substr` can be found. `Upto(chars, str)` generates the positions within string `str` at which any character in `chars` appears. (The initial argument to `find` is a string, delimited by double quote marks; the initial argument to `upto` is a `cset` [character set], delimited by single quote marks.) The prefix operator `!` generates all elements of its operand, which can be a string, list, record, file, or table.

EXAMPLE 6.86
A generator inside an expression

In comparison to conventional iterators, however, the generators of Icon are more deeply embedded in the semantics of the language. A generator can be used in any context that expects an expression. The larger context is then capable of generating multiple results. The following code will print all positions in `s` that follow a blank:

```
every i := 1 + upto(' ', s) do {
    write(i)
}
```

This can even be written as

```
every write(1 + upto(' ', s))
```
■

Generators in Icon are used not only for iteration, but also for *goal-directed search*, implemented via *backtracking*. (Backtracking is also fundamental to Prolog, which we will study in Chapter 11.) Where most languages use Boolean expressions to control selection and logically controlled loops, Icon uses a more general notion of *success* and *failure*. A conditional statement such as

EXAMPLE 6.87
Generating in search of success

```
if 2 < 3 then {
    ...
}
```

is said to execute not because the condition `2 < 3` is true, but because the *comparison* `2 < 3` *succeeds*. The distinction is important for generators, which are capable of producing results repeatedly until one of them causes the surrounding context to succeed (or until no more results can be produced). For example, in

```
if (i := find("abc", s)) > 6 then {
    ...
}
```

the body of the `if` statement will be executed only if the string `"abc"` appears beyond the sixth position in `s`. Because `find` generates its results in order, `i` will represent the first such position (if any). The execution model is as follows: `find` is capable of generating all positions at which `"abc"` occurs in `s`. Suppose the first such occurrence is at position 2. Then `i` is assigned the value 2, but the comparison `2 > 6` fails. Because there is a generator inside the failed expression, Icon will resume that generator and reevaluate the expression for the next generated value. It will continue this reevaluation process until the comparison succeeds, or until the generator runs out of values, in which case it (the generator) fails, the overall expression fails definitively, and the body of the `if` is skipped. ■

EXAMPLE 6.88
Backtracking with multiple generators

If a failed expression contains more than one generator, all possible values will be explored systematically. The body of the following `if`, for example, will be executed if and only if an `x` appears at the same position in both `s` and `t`, with `i` denoting the first such matching position:

```
if (i := find("x", s)) = find("x", t) then {
    ...
}
```

If there is no matching position, then `i` will be set to the position of the final `x` in `s`, but the body of the loop will be skipped. If the programmer wishes to avoid changing `i` in the case where the overall test fails, then the *reversible assignment* operator, `<-` can be used instead of `:=`. When Icon backtracks past a reversible assignment, it restores the original value. ■

Any user-defined subroutine in Icon can be a generator if it uses the `suspend` *expr* statement instead of `return` *expr*. `Suspend` is Icon's equivalent of `yield`. If the expression following `suspend` contains an invocation of a generator, then the subroutine will suspend repeatedly, once for each generated value.

✓ **CHECK YOUR UNDERSTANDING**

44. Explain how Icon generators differ from the iterators of Clu, Python, Ruby, and C#, and from the iterator objects of Euclid, C++, and Java.
45. Describe the notions of *success* and *failure* in Icon.
46. What is *backtracking*? Why is it useful?
47. Name a language other than Icon in which backtracking plays a fundamental role.

6 Control Flow

6.7 Nondeterminacy

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

Dijkstra [Dij75] has 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, which we will study in more detail in Chapter 12. 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 difficultto 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:

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
```

```
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.

Nondeterministic Concurrency

EXAMPLE 6.92
Nondeterministic message receipt

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 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 ◎6.7). ■

EXAMPLE 6.93
Nondeterministic server in SR

Most message-based concurrent languages provide at least one nondeterministic construct that can be used to specify communication with any of several possible communication partners. 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)
```

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.

```
# local subroutines:
proc lookup ...         # find info in dictionary
proc update ...         # change info in dictionary
# 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 12.5.3 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 on page 119). ■

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

DESIGN & IMPLEMENTATION

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 retain 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 probably don't 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.

that is true. A program may therefore produce unexpected or even unpredictable 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

48. What is a *guarded command*?
49. Explain why nondeterminacy is particularly important for concurrent programs.
50. Give three alternative definitions of *fairness* in the context of nondeterminacy.
51. 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

6.34 (David Hanson [Han93].) Write a program in Icon that will print the *k* most common words in its input, one per line, with each preceded by a count of the number of times it appears. If parameter *k* is not specified on the command line, use 10 by default. You will want to consult the Icon manual (available on-line [GG96]). In addition to `suspend`, `upto`, and `write`, discussed here, you may find it helpful to learn about `integer`, `many`, `pull`, `read`, `sort`, `table`, and `tab`. When fed the Gettysburg Address, your program should print:

```
13  that
9   the
8   we
8   to
8   here
7   a
6   and
5   of
5   nation
5   have
```

6.35 Write a `findRE` generator in Icon that mimics the behavior of `find`, but takes as its first parameter a regular expression. Use a string to represent your regular expression, with syntax as in Section 2.1.1. Use empty parentheses to represent ϵ. Give highest precedence to Kleene closure, then concatenation, then alternation. You may assume that we never search for vertical bar, asterisk, or parenthesis characters.

6.36 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
```

6.37 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

6.43 Learn about Snobol, an earlier language by Ralph Griswold, who also designed Icon. How do the two languages compare?

6.44 Chapter 18 of Griswold's text on Icon [GG96] discusses scanning and parsing. After reading this chapter, explain how backtracking search can be used to generalize recursive descent. What classes of grammars can you parse with this generalized technique? What is the worst-case time complexity?

6.45 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 ◎6.93?

6.46 Explain how to use threads in Java to achieve the effect of Example ◎6.93.

7 Data Types

7.2.4 The ML Type System

EXAMPLE 7.96
Fibonacci function in ML

The following is an ML version of the tail-recursive Fibonacci function introduced in Section 6.6.1:

```
1. fun fib (n) =
2.     let fun fib_helper (f1, f2, i) =
3.         if i = n then f2
4.         else fib_helper (f2, f1+f2, i+1)
5.     in
6.         fib_helper (0, 1, 0)
7.     end;
```

The `let` construct introduces a nested scope: function `fib_helper` is nested inside `fib`. The body of `fib` is the expression `fib_helper (0, 1, 0)`. The body of `fib_helper` is an `if`... `then` ... `else` expression; it evaluates to either `f2` or to `fib_helper (f2, f1+f2, i+1)`, depending on whether the third argument to `fib_helper` is `n` or not.

Given this function definition, an ML compiler will reason roughly as follows: Parameter `i` of `fib_helper` must have type `int`, because it is added to 1 at line 4. Similarly, parameter `n` of `fib` must have type `int`, because it is compared to `i` at line 3. In the specific call to `fib_helper` at line 6, the types of all three arguments are `int`, so in this context at least, the types of `f1` and `f2` are `int`. Moreover the type of `i` is consistent with the earlier inference, namely `int`, and the types of the arguments to the recursive call at line 4 are similarly consistent. Since `fib_helper` returns `f2` at line 3, the result of the call at line 6 will be an `int`. Since `fib` immediately returns this result as its own result, the return type of `fib` is `int`. ■

EXAMPLE 7.97
Expression types

Because ML is a functional language, every construct in ML is an expression. The ML type system infers a type for every object and every expression. Because functions are first-class values, they too have types. The type of `fib` above is `int -> int`; that is, a function from integers to integers. The type of `fib_helper` is int * int * int -> int; that is, a function from integer triples to integers. In denotational terms, int * int * int is a three-way Cartesian product. ■

Type correctness in ML amounts to what we might call type *consistency*: a program is type correct if the type checking algorithm can reason out a unique type for every expression, with no contradictions and no ambiguous occurrences of overloaded names. (For built-in arithmetic and comparison operators, ML assumes that arguments are integers if it cannot determine otherwise. Haskell is a bit more general: it allows the arguments to be of any type that supports the required operations.) If the programmer uses an object inconsistently, the compiler will complain. In a program containing the following expressions,

EXAMPLE 7.98
Type inconsistency

```
fun circum (r) = r * 2.0 * 3.14159;
...
circum (7)
```

the compiler will infer that circum's parameter is of type real, and will then complain when we attempt to pass an integer argument. ■

Though usually compiled instead of interpreted, ML is intended for interactive use. The programmer interacts with the ML system "on-line," giving it input a line at a time. The system compiles this input incrementally, binding machine language fragments to function names, and producing any appropriate compile-time error messages. This style of interaction blurs the traditional distinction between interpretation and compilation, but has more of the flavor of the latter. The language implementation remains active during program execution, but it does not actively manage the execution of program fragments: it *transfers* control to them and waits for them to return.

In comparison to languages in which programmers must declare all types explicitly, ML's type inference system has the advantage of brevity and convenience for interactive use. More important, it provides a powerful form of implicit parametric polymorphism more or less for free. While all uses of objects in an ML program must be consistent, they do not have to be completely specified:

EXAMPLE 7.99
Polymorphic functions

```
fun compare (x, p, q) =
    if x = p then
        if x = q then "both"
        else "first"
    else
        if x = q then "second"
        else "neither";
```

Here the equality test (=) is a built-in polymorphic function of type 'a * 'a -> bool; that is, a function that takes a pair of arguments of the same type and produces a Boolean result. The token 'a is called a *type variable*; it stands for any type, and takes, implicitly, the role of an explicit type parameter in a generic construct (Sections 8.4 and 9.4.4). Every instance of 'a in a given call to = must represent the same type, but instances of 'a in different calls can be different. Starting with the type of =, an ML compiler can reason that the type of `compare` is `'a * 'a * 'a -> string`. Thus `compare` is polymorphic; it does not depend on the types of `x`, `p`, and `q`, so long as they are all the same. The key point to observe is that the programmer did not have to do anything special to make `compare` polymorphic: polymorphism is a natural consequence of ML-style type inference. ■

Type Checking

An ML compiler verifies type consistency with respect to a well-defined set of constraints. Specifically,

- All occurrences of the same identifier (subject to scope rules) must have the same type.
- In an `if`... `then` ... `else` expression, the condition must be of type `bool`, and the `then` and `else` clauses must have the same type.
- A programmer-defined function has type `'a -> 'b`, where `'a` is the type of the function's parameter, and `'b` is the type of its result. As we shall see shortly, all functions have a single parameter. One obtains the *appearance* of multiple parameters by passing a *tuple* as argument.
- When a function is applied (called), the type of the argument that is passed must be the same as the type of the parameter in the function's definition. The type of the application (call) is the same as the type of the result in the function's definition.

In any case where two types *A* and *B* must be "the same," the ML compiler must *unify* what it knows about *A* and *B* to produce a (potentially more detailed) description of their common type. For example, if the compiler has determined that `E1` is an expression of type `'a * int` (that is, a two-element tuple whose second element is known to be an integer), and that `E2` is an expression of type `string * 'b`, then in the expression `if x then E1 else E2`, it can infer that `'a` is `string` and `'b` is `int`. Thus `x` is of type `bool`, and `E1` and `E2` are of type `string * int`.

Lists

As in most functional languages, ML programmers tend to make heavy use of lists. In languages like Lisp and Scheme, which are dynamically typed (and also

DESIGN & IMPLEMENTATION

Unification

Unification is a powerful technique. In addition to its role in type inference (which also arises in the templates [generics] of C++), unification plays a central role in the computational model of Prolog and other logic languages. We will consider this latter role in Section 11.1. In the general case the cost of unifying the types of two expressions can be exponential [Mai90], but the pathological cases tend not to arise in practice.

EXAMPLE 7.100
Polymorphic list operators

implicitly polymorphic), lists may contain objects of arbitrary types. In ML, all elements of a given list must have the same type, but—and this is important—functions that manipulate lists without performing operations on their members can take any kind of list as argument:

```
fun append (l1, l2) =
    if l1 = nil then l2
    else hd (l1) :: append (tl (l1), l2);
fun member (x, l) =
    if l = nil then false
    else if x = hd (l) then true
    else member (x, tl (l));
```

Here append is of type `'a list * 'a list -> 'a list`; `member` is of type `'a * 'a list -> bool`. The reserved word `nil` represents the empty list. The built-in `::` constructor is analogous to `cons` in Lisp. It takes an element and a list and tacks the former onto the beginning of the latter; its type is `'a * 'a list -> 'a list`. The `hd` and `tl` functions are analogous to `car` and `cdr` in Lisp. They return the head and the remainder, respectively, of a list created by `::`. ■

EXAMPLE 7.101
List notation

Lists are most often written in ML using "square bracket" notation. The token `[]` is the same as `nil`. `[A, B, C]` is the same as `A :: B :: C :: nil`. Only "proper" lists—those that end with nil—can be represented with square brackets. The append function defined above is actually provided in ML as a built-in infix constructor, `@`. The expression `[a, b, c] @ [d, e, f, g]` evaluates to `[a, b, c, d, e, f, g]`. ■

Since ML lists are homogeneous (all elements have the same type), one might wonder about the type of `nil`. To allow it to take on the type of any list, `nil` is defined not as an object, but as a built-in polymorphic function of type `unit -> 'a list`. The built-in type `unit` is simply a placeholder, analogous to `void` in C. A function that takes no arguments is said to have a parameter of type `unit`. A function that is executed only for its side effects (ML is not purely functional) is said to return a result of type `unit`.

Overloading

We have already seen that the equality test (`=`) is a built-in polymorphic operator. The same is not true of ordering tests (`<`, `<=`, `>=`, `>`) or arithmetic operators (`+`, `-`, `*`). The equality test can be defined as a polymorphic function because it accepts arguments of *any* type. The relations and arithmetic operators work only on certain types. To avoid limiting them to a single type of argument (e.g., integers), ML defines them as overloaded names for a collection of built-in functions, each of which operates on objects of a different type (integers, floating-point numbers, strings, etc.). The programmer can define additional such functions for new types.

EXAMPLE 7.102
Resolving ambiguity with explicit types

Unfortunately, overloading sometimes interferes with type inference—there may not be enough information in an otherwise valid program to resolve which function is named by an overloaded operator:

```
fun square (x) = x * x;
```

Here the ML compiler cannot tell whether * is meant to refer to integer or floating-point multiplication; it assumes the former by default. If this is not what the programmer wants, the alternative must be specified explicitly:

```
fun square (x : real) = x * x;
```
■

In addition to allowing the resolution of overloaded symbols, explicit type declarations serve as "verified documentation" in ML programs. ML programmers often declare types for variables even when they aren't required, because the declarations make a program easier to read and understand. Readability could also be enhanced by comments, of course, but programmer-specified types have a very important advantage: the compiler understands their meaning, and ensures that all uses of an object are consistent with its declared type.

EXAMPLE 7.103
Type classes in Haskell

Haskell adopts a more general approach to overloading known as *type classes*. The equality functions, for example, are declared (but not defined) in a predefined class Eq:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool    -- type signature
    x /= y     =  not (x == y)      -- default implementation of /=
```

Here a (written without a tick mark) is a type parameter: both == and /= take two parameters of the same type and return a Boolean result. Any value that is passed to one of these functions will be inferred to be of some type in class Eq. Any value that is passed to one of the ordering functions (<, <=, >=, >) will similarly be inferred to be of some type in class Ord:

```
class (Eq a) => Ord a where
    (<), (<=), (>=), (>) :: a -> a -> Bool
    max, min             :: a -> a -> a
```

The "(Eq a) =>" in the header of this declaration indicates that Ord is an extension of Eq; every type in class Ord must support the operations of class Eq as well. There is a strong analogy between type classes and the *interfaces* of languages with mix-in inheritance (Section ◎9.5.4). ■

Pattern Matching

In our discussion so far, we have been "glossing over" another key feature of ML and its types: namely, pattern matching. One of the simplest forms of pattern matching occurs in functions of more than one parameter. Strictly speaking, such functions do not exist. Every function in ML takes a *single* argument, but this argument may be a *tuple*. A tuple resembles the records (structures) of many other languages, except that its members are identified by position, rather than by name. As an example, the function compare defined above takes a three-element tuple as argument. All of the following are valid:

EXAMPLE 7.104
Pattern matching of argument tuples

```
compare (1, 2, 3);
let val t = ("larry", "moe", "curly") in compare (t) end;
let val d = (2, 3) in
    let val (a, b) = d in
        compare (1, a, b)
    end
end;
```

Here pattern matching occurs not only between the parameters and arguments of the call to `compare`, but also between the left- and right-hand sides of the `val` construct. (The reserved word `val` serves to declare a name. The construct `fun inc (n) = n+1;` is syntactic sugar for `val inc = (fn n => n+1);`.) ■

EXAMPLE 7.105
Swap in ML

As a somewhat more plausible example, we can define a highly useful function that reverses a two-element tuple:

```
fun swap (a, b) = (b, a);
```

Since ML is (mostly) functional, swap is not intended to exchange the value of objects; rather, it takes a two-element tuple as argument, and produces the symmetrical two-element tuple as a result. ■

Pattern matching in ML works not only for tuples, but for any built-in or user-defined *constructor* of composite values. Constructors include the parentheses used for tuples, the square brackets used for lists, several of the built-in operators (`::`, `@`, etc.), and user-defined constructors of `datatypes` (see below). Literal constants are even considered to be constructors, so the tuple `t` can be matched against the pattern `(1, x)`: the match will succeed only if `t`'s first element is `1`.

In a call like `compare (t)` or `swap (2, 3)`, an ML implementation can tell at compile time that the pattern match will succeed: it knows all necessary information about the structure of the value being matched against the pattern. In other cases, the implementation can tell that a match is doomed to fail, generally because the types of the pattern and the value cannot be unified. The more interesting cases are those in which the pattern and the value have the same type (i.e., could be unified), but the success of the match cannot be determined until run time. If `l` is of type `int list`, for example, then an attempt to "deconstruct" `l` into its head and tail may or may not succeed, depending on `l`'s value:

EXAMPLE 7.106
Run-time pattern matching

```
let val head :: rest = l in ...
```

If `l` is `nil`, the attempted match will produce an *exception* at run time (we will consider exceptions further in Section 8.5). ■

EXAMPLE 7.107
ML `case` expression

We have seen how pattern matching works in function calls and `val` constructs. It is also supported by a `case` expression. Using `case`, the `append` function above could have been written as follows:

```
fun append (l1, l2) =
    case l1 of
        nil => l2
      | h :: t => h :: append (t, l2);
```

Here the code generated for the case expression will pattern-match l1 first against nil and then against h :: t. The case expression evaluates to the subexpression following the => in the first arm whose pattern matches. The compiler will issue a warning message at compile time if the patterns of the arms are not exhaustive, or if the pattern in a later arm is completely covered by one in an earlier arm (implying that the latter will never be chosen). ■

A useless arm is probably an error, but harmless, in the sense that it will never result in a dynamic semantic error message. Nonexhaustive cases may be intentional, if the programmer can predict that the pattern will always work at run time. Our append function would have generated such a warning if written as follows:

EXAMPLE 7.108
Coverage of case labels

```
fun append (l1, l2) =
    if l1 = nil then l2
    else let val h::t = l1 in h :: append (t, l2) end;
```

Here the compiler is unlikely to realize that the let construct in the else clause will be elaborated only if l1 is nonempty. (This example looks easy enough to figure out, but the general case is uncomputable, and most compilers won't contain special code to recognize easy cases.) ■

EXAMPLE 7.109
Function as a series of alternatives

When the body of a function consists entirely of a case expression, it can also be written as a simple series of alternatives:

```
fun append (nil, l2) = l2
 |  append (h::t, l2) = h :: append (t, l2);
```
■

Pattern matching features prominently in other languages as well, particularly those (such as Snobol, Icon, and Perl) that place a heavy emphasis on strings. ML-style pattern matching differs from that of string-oriented languages in its integration with static typing and type inference. Snobol, Icon, and Perl are all dynamically typed.

By casting "multiargument" functions in terms of tuples, ML eliminates the asymmetry between the arguments and return values of functions in many other languages. As shown by swap above, a function can return a tuple just as easily as it can take a tuple argument. Pattern matching allows the elements of the tuple to be extracted by the caller:

EXAMPLE 7.110
Pattern matching of return tuple

```
let val (a, b) = swap (c, d) in ...
```

Here a will have the value given by d; b will have the value given by c. ■

Datatype ***Constructors***

In addition to lists and tuples, ML provides built-in constructors for records, together with a datatype mechanism that allows the programmer to introduce other kinds of composite types. A record is a composite object in which the elements have names, but no particular order (the language implementation must choose an order for its internal representation, but this order is not visible to the programmer). Records are specified using a "curly brace" constructor: `{name = "Abraham Lincoln", elected = 1860}`. (The same value can be denoted `{elected = 1860, name = "Abraham Lincoln"}`.) ■

EXAMPLE 7.111 ML records

EXAMPLE 7.112 ML `datatypes`

ML's `datatype` mechanism introduces a type name and a collection of constructors for that type. In the simplest case, the constructors are all functions of zero arguments, and the type is essentially an enumeration:

```
datatype weekday = sun | mon | tue | wed | thu | fri | sat;
```

In more complicated examples, the constructors have arguments, and the type is essentially a union (variant record):

```
datatype yearday = mmdd of int * int | ddd of int;
```

This code defines `mmdd` as a constructor that takes a pair of integers as argument, and `ddd` as a constructor that takes a single integer as argument. The intent is to allow days of the year to be specified either as (month, day) pairs or as integers in the range 1 . . 366. In a non–leap year, the Fourth of July could be represented either as `mmdd (7, 4)` or as `ddd (188)`, though the equality test `mmdd (7, 4) = ddd (188)` would fail unless we made `yearday` an *abstract* type (similar to the Euclid module types of Section 3.3.4), with its own, special, equality operation. ■

EXAMPLE 7.113 Recursive `datatypes`

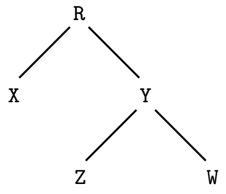
ML's `datatypes` can even be used to define recursive types, without the need for pointers. The canonical ML example is a binary tree:

```
datatype int_tree = empty | node of int * int_tree * int_tree;
```

By introducing an explicit type variable in the definition, we can even create a generic tree whose elements are of any homogeneous type:

```
datatype 'a tree = empty | node of 'a * 'a tree * 'a tree;
```

Given this definition, the tree



can be written `node (#"R", node (#"X", empty, empty), node (#"Y", node (#"Z", empty, empty), node (#"W", empty, empty)))`. Recursive types also appear in Lisp, Clu, Java, C#, and other languages with a reference model of variables; we will discuss them further in Section 7.7. ■

EXAMPLE 7.114 Type equivalence in ML

Because of its use of type inference, ML generally provides the effect of structural type equivalence. Definitions of `datatypes` can be used to obtain the effect of name equivalence when desired:

```
datatype celsius_temp = ct of int;
datatype fahrenheit_temp = ft of int;
```

A value of type `celsius_temp` can then be obtained by using the `ct` constructor:

```
val freezing = ct (0);
```

Unfortunately, `celsius_temp` does not automatically inherit the arithmetic operators and relations of `int`: unless the programmer defines these operators explicitly, the expression `ct (0) < ct (20)` will generate an error message along the lines of "operator not defined for type." ■

✓ CHECK YOUR UNDERSTANDING

54. Under what circumstances does an ML compiler announce a type clash?
55. Explain how the type inference of ML leads naturally to polymorphism.
56. What is a *type variable*? Give an example in which an ML programmer might use such a variable explicitly.
57. How do lists in ML differ from those of Lisp and Scheme?
58. Why do ML programmers often declare the types of variables, even when they don't have to?
59. What is *unification*? What is its role in ML?
60. List three contexts in which ML performs *pattern matching*.
61. Explain the difference between *tuples* and *records* in ML. How does an ML record differ from a record (structure) in languages like C or Pascal?
62. What are ML `datatype`s? What features do they subsume from imperative languages such as C and Pascal?

7 Data Types

7.3.3 `With` Statements

The Pascal `with` statement introduces a nested scope in which the fields of the named record become visible as if they were ordinary variables. The record is said to be *opened*. As shown in Figure ©3.19 (page ©32), a `with` statement can be implemented within the compiler by pushing an entry that represents the record type onto the symbol table scope stack.

EXAMPLE 7.115
Elliptical references in Cobol and PL/I

`With` statements are a formalization of the *elliptical references* of Cobol and PL/I, a language feature that permits portions of a fully qualified name to be omitted if no ambiguity results. In the Cobol equivalent of Example 7.46, `name of elements(1) of chemical_composition of ruby` could probably be abbreviated `name of elements(1) of ruby`, since `ruby` is unlikely to have a field named `elements` within anything other than its `chemical_composition` field. The rest of the reference is required, however, if there is another record of the same type as `ruby` in the current scope, and if the `elements` array within that type contains more than a single element. ■

EXAMPLE 7.116
Pascal `with` statement (reprise)

Elliptical references can be difficult to read, since they rely implicitly on the uniqueness of field names. A `with` statement specifies the elided information more explicitly, making misunderstandings less likely. Repeating example 7.46:

```
with ruby.chemical_composition.elements[1] do begin
    name := 'Al';
    atomic_number := 13;
    atomic_weight := 26.98154;
    metallic := true
end;
```
■

The `with` statements of Pascal still suffer from problems, however:

1. There is no easy way to manipulate fields of two records of the same type simultaneously (e.g., to copy some of the fields of one into corresponding

fields of the other). A `with` statement can be used to open one of the records, but not the other.

2. Naming conflicts arise if any of the fields of an opened record have the same name as local objects. Since the `with` statement is a nested scope, the local objects become temporarily inaccessible.
3. In a long `with` statement, or in nested `with` statements that open records of different types, the correspondence between field names and the records to which they belong can become unclear.

EXAMPLE 7.117
Modula-3 `with` statement

Modula-3 and Fortran 2003 address these problems by redefining the `with` statement in a more general form. Rather than opening a record, a Modula-3 `WITH` statement or Fortran `associate` construct introduces one or more aliases for complicated expressions. Recasting our example in the style of Modula-3, we can write:

```
WITH e = ruby.chemical_composition.elements[1] DO
    e.name := "Al";
    e.atomic_number := 13;
    e.atomic_weight := 26.98154;
    e.metallic := true;
END;
```

Here `e` is an alias for `ruby.chemical_composition.elements[1]`. The fields of the record are not *directly* visible, but they can be accessed easily, simply by prepending '`e.`' to their names. ■

EXAMPLE 7.118
Multiple-object `with` statements

To access more than one record at a time, one can write

```
WITH e = whatever, f = whatever DO
    e.field1 := f.field1;
    e.field3 := f.field3;
    e.field7 := f.field7;
END;
```
■

EXAMPLE 7.119
Non-record `with` statements

Modula-3 `WITH` statements and Fortran `associate` constructs can even be used to create aliases for objects other than records, e.g., to test and then use a complicated expression without writing it out twice:

DESIGN & IMPLEMENTATION

`With` statements

A compiler generally implements `with` or `associate` statements by creating a hidden pointer to the opened or aliased record. All uses of the record inside the `with` statement access fields efficiently via offsets from the hidden pointer. Equivalent efficiency can usually be achieved without the `with` statement, but only if the complier implements global common subexpression analysis, a non-trivial form of code improvement that we defer to Section 16.4.

```
WITH d = complicated_expression DO
    IF d # 0.0 THEN val := n/d ELSE val := 0.0 END;
END;
```

■

EXAMPLE 7.120
Emulating `with` in Scheme

A similar effect, of course, can be achieved in many languages without a special construct. Most functional languages, for example, include a `let` statement that introduces a nested scope. In Scheme:

```
(let ((d  complicated_expression))
  (if (not (= d 0)) (/ n d) 0))
```

This code has roughly the same effect as the code in Example ◎7.119. Example ◎7.118, which copies fields of `f` into `e`, would require the use of nonfunctional language features. ■

EXAMPLE 7.121
Emulating `with` in C

In C one might write

```
{                                         {
    my_struct *e = &whatever;                 double d = complicated_expression;
    my_struct *f = &whatever;                 val = (d ? n/d : 0);
    e->field1 = f->field1;                }
    e->field3 = f->field3;
    e->field7 = f->field7;
}
```

This code depends on the ability of the C programmer to declare variables in nested blocks and to create pointers to nonheap objects. Pascal does not permit nested declarations; neither Pascal nor Modula-3 permits the nonheap pointers. Reference types in C++, which we will introduce in Section 8.3.1, can be used in place of pointers in the C example to produce an even closer approximation of the Modula-3 or Fortran syntax. ■

✓ CHECK YOUR UNDERSTANDING

63. What is a `with` statement? What purpose does it serve?
64. What are *elliptical references*?
65. What are the limitations of `with` statements as realized in Pascal? How are these limitations overcome in Modula-3 and Fortran 2003? How are they avoided in languages like Scheme?
66. Explain how to emulate the behavior of `with` statements in languages like C.
67. Is a `with` statement purely a notational convenience, or does it have pragmatic implications as well?

7 Data Types

7.3.4 Variant Records (Unions)

EXAMPLE 7.122
Variant record in Pascal

A variant record provides two or more alternative fields or collections of fields, only one of which is valid at any given time. Building on the `element` type of Example 7.36, we might write the following in Pascal.

```
type long_string = packed array [1..200] of char;
type string_ptr = ^long_string;
type element = record
    name : two_chars;
    atomic_number : integer;
    atomic_weight : real;
    metallic : Boolean;
    case naturally_occurring : Boolean of
      true : (
        source : string_ptr;
            (* textual description of principal commercial source *)
        prevalence : real;
            (* fraction, by weight, of Earth's crust *)
      );
      false : (
        lifetime : real;
            (* half-life in seconds of most stable known isotope *)
      )
end;
```

Here the `naturally_occurring` field of the record is known as its *tag*, or *discriminant*. It's a field that indicates which variant is valid. In this example, a `true` tag indicates that the element has at least one naturally occurring stable isotope; in this case the record contains two additional fields—`source` and `prevalence`—that describe how the element may be obtained and how commonly it occurs.

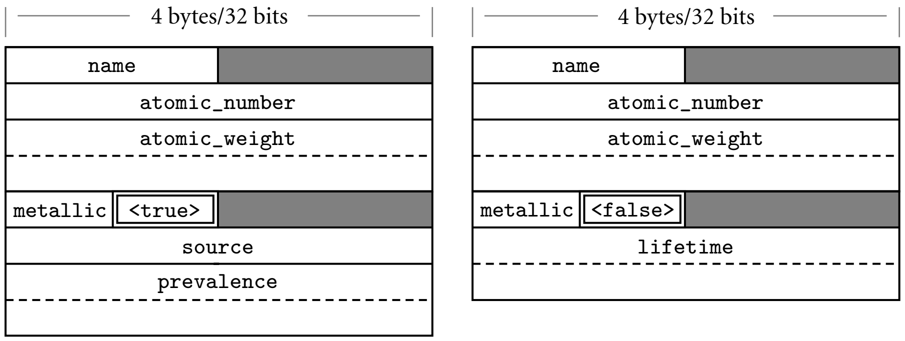


Figure 7.15 Likely memory layouts for `element` variants. The value of the `naturally_occurring` field (shown here with a double border) determines which of the interpretations of the remaining space is valid. Type `string_ptr` is assumed to be represented by a (4-byte) pointer to dynamically allocated storage.

A `false` tag indicates that the element results only from atomic collisions or the decay of heavier elements; in this case, the record contains an additional field—`lifetime`—that indicates how long atoms so created tend to survive before undergoing radioactive decay. Each of the parenthesized field lists (one containing `source` and `prevalence`, the other containing `lifetime`) is known as a *variant*. Either the first or the second variant may be useful, but never both at once. From an implementation point of view, these nonoverlapping uses mean that the variants may share space (see Figure 7.15). ■

EXAMPLE 7.123
Fortran `equivalence` statement

Variant records have their roots in the `equivalence` statement of Fortran I and in the `union` types of Algol 68. The Fortran syntax looks like this:

```
integer i
real r
logical b
equivalence (i, r, b)
```

The `equivalence` statement informs the compiler that `i`, `r`, and `b` will never be used at the same time, and should share the same space in memory. ■

EXAMPLE 7.124
Mixing structs and unions in C

Pascal's principal contribution to union types (retained by Modula and Ada) was to integrate them with records. This was an important contribution, because the need for alternative types seldom arises anywhere else. In our running example, we use the same field-name syntax to access both the `atomic_weight` and `lifetime` fields of an `element`, despite the fact that the former is present in every `element`, while the latter is present only in those that are not naturally occurring. Without the integration of records and unions, the notation is less convenient. Here's what it looks like in C:

```
struct element {
    char name[2];
    int atomic_number;
    double atomic_weight;
    _Bool metallic;
    _Bool naturally_occurring;
    union {
        struct {
            char *source;
            double prevalence;
        } natural_info;
        double lifetime;
    } extra_fields;
} copper;
```

Because the `union` is not a part of the `struct`, we have to introduce two extra levels of naming. The third field is still `copper.atomic_weight`, but the `source` field must be accessed as `copper.extra_fields.natural_info.source`. A similar situation occurs in ML, in which `datatype`s can be used for unions, but the notation is not integrated with records (Exercise ◎7.27). ■

Safety

EXAMPLE 7.125
Breaking type safety with `equivalence`

One of the principal problems with `equivalence` statements is that they provide no built-in means of determining which of the `equivalence`-ed objects is currently valid: the program must keep track. Mistakes in which the programmer writes to one object and then reads from the other are relatively common:

```
r = 3.0
...
print '(I10)', i
```

Here the `print` statement, which attempts to output `i` as a 10-digit 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. ■

Fortran `equivalence` statements introduce an extreme case of aliases: not only are there two names for the "same thing" (in this case the same block of storage), but the types associated with those names are different. To address this potential source of bugs, the Algol 68 designers required that the language implementation track `union`-ed types at run time:

EXAMPLE 7.126
Union conformity in Algol 68

```
union (int, real, bool) uirb
    # uirb can be an integer, a floating-point number, or a Boolean #
...
uirb := 1        # uirb is now an integer #
...
uirb := 3.14     # uirb is now a floating-point number #
```

To use the value stored inside a union, the programmer must employ a special form of `case` statement (called a *conformity clause* in Algol 68) that determines which type is currently valid:

```
case uirb in
    (int i) : print(i),
    (real r) : print(r),
    (bool b) : print(b)
esac
```

The labels on the arms of the `case` statement provide names for the "deunified" values. A similar `tagcase` construct can be found in Clu. ■

To enforce correct usage of union types in Algol 68, the language implementation must maintain a hidden field for every union object that indicates which type is currently valid. When an object of a union type is assigned a value, the hidden field is also set, to indicate the type of the value just assigned. When execution encounters a conformity clause, the hidden field is inspected to determine which arm to execute.

EXAMPLE 7.127
Tagged variant record in Pascal

In effect, the tag field of a Pascal variant record is an explicit representation of the hidden field required in an Algol 68 union. Our integer/floating-point/Boolean example could be written as follows in Pascal.

```
type tag = (is_int, is_real, is_bool);
var uirb : record
    case which : tag of
        is_int : (i : integer);
        is_real : (r : real);
        is_bool : (b : Boolean)
end;
```
■

EXAMPLE 7.128
Breaking type safety with variant records

Unfortunately, while the hidden tag of an Algol 68 union can only be changed implicitly, by assigning a value of a different type to the union as a whole, the tag of a Pascal variant record can be changed by an ordinary assignment statement. The compiler can generate code to verify that a field in variant *v* is never accessed unless the value of the tag indicates that *v* is currently valid, but this is not enough to guarantee type safety. It can catch errors of the form

```
uirb.which := is_real;
uirb.r := 3.0;
...
writeln(uirb.i);    (* dynamic semantic error *)
```

but it cannot catch the following:

```
uirb.which := is_real;
uirb.r := 3.0;
uirb.which := is_int;
...                 (* no intervening assignment to i *)
writeln(uirb.i);    (* ouch! *)
```

Any Pascal implementation will accept this code, but the output is likely to be erroneous, just as it was in Fortran. ■

Semantically speaking, changing the tag of a Pascal variant record should make the remaining fields of the variant *uninitialized*. It is possible, by adding hidden fields, to flag them as such and generate a semantic error message on any subsequent access, but the code to do so is expensive [FL80], and outlaws programs which, while arguably erroneous, are permitted by the language definition (Exercise ◎7.33).

EXAMPLE 7.129
Untagged variants in Pascal

The situation in Pascal is actually worse than our example so far might imply. Additional insecurity stems from the fact that Pascal's tag fields are *optional*. We could drop the which field of our uirb record:

```
var uirb : record
    case tag of
        is_int : (i : integer);
        is_real : (r : real);
        is_bool : (b : Boolean)
end;
...
uirb.r := 3.0;
...                 (* no intervening assignment to i *)
writeln(uirb.i);    (* ouch! *)
```

Now the language implementation is not required to devote any space to either an explicit or hidden tag, but even the limited form of checking (make sure the tag has an appropriate value when a field of a variant is accessed) is no longer possible (but see Exercise ◎7.34). Variant records with tags (explicit or hidden) are known as *discriminated unions*. Variant records without tags are known as *nondiscriminated unions*. ■

The degree of type safety provided is arguably the most important dimension of variation among the variant records and union types of modern languages. Though designed after Algol 68 (and borrowing its union terminology), the union types of C are semantically closer to Fortran's equivalence statements. Their fields share space, but nothing prevents the programmer from using them in inappropriate ways. By contrast, the variant records of Ada are syntactically similar to those of Pascal, but are as type-safe as the unions of Algol 68. Concerned at the lack of type safety in Pascal and Modula-2, and reluctant to introduce the complexity of Ada's rules, the designers of Modula-3 chose to eliminate variant records from the language entirely.

Variants in Ada

EXAMPLE 7.130
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., A := {which => is_real, r => pi};). 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 : real;
    metallic : Boolean;
    case naturally_occurring is
        when true =>
            source : string_ptr;
            prevalence : real;
        when false =>
            lifetime : real;
    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 7.131
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 7.132
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:

```
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 : real;
end record;
```

The `<>` notation in the initial definition of `element_array` indicates that the bounds are not statically known. We will have more to say about dynamic arrays in Section 7.4.2. 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). ■

DESIGN & IMPLEMENTATION

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 has a similar rule, designed for a similar purpose. When a variant record is allocated from the heap in Pascal (via the built-in `new` operator), the programmer has the option of specifying `case` labels for the variant portions of the record. A record so allocated is never allowed to change to a different variant, so the implementation can allocate precisely the right amount of space.

Modula-2, which does not provide `new` as a built-in operation, eliminates the ordering restriction on variants. All variables of a variant record type must be large enough to hold any variant. The usual implementation assigns a fixed offset to every field, with holes following small internal variants as necessary (see Figure ◎7.16 and Exercise ◎7.35).

```
TYPE element = RECORD
    name : ARRAY [1..2] OF CHAR;
    metallic : BOOLEAN;
    CASE naturally_occurring : BOOLEAN OF
        TRUE :
            source : string_ptr;
            prevalence : REAL;
      | FALSE :
            lifetime : REAL;
    END;
    atomic_number : INTEGER;
    atomic_weight : REAL;
END;
```

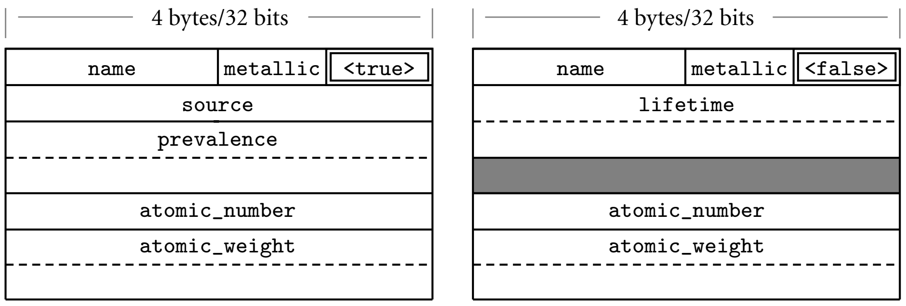


Figure 7.16 Likely memory layout for a variant record in Modula-2. Here the variant portion of the record is not required to lie at the end. Every field has a fixed offset from the beginning of the record, with internal holes as necessary following small-size variants.

The Object-Oriented Alternative

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, replaces variants with a more general mechanism for *type extension* (Section 9.2.4), and the designers of Java and C# dropped the unions of C and C++. In place of the C code of Example 7.124, a Java programmer might write

EXAMPLE 7.133
Derived types as an alternative to unions

```
class Element {
    public String name;
    public int atomic_number;
    public double atomic_weight;
    public boolean metallic;
}
class NaturalElement extends Element {
    public String source;
    public double prevalence;
}
```

```
class SyntheticElement extends Element {
    public double lifetime;
}
```

Like the unification of records and variants of Pascal, this approach avoids the artificial `extra_fields` and `natural_info` names of Example ◎7.124. Like the discriminated subtypes of Ada, however, it constrains each variable to a single variant at elaboration time; this cannot be changed by subsequent assignment. ■

✓ CHECK YOUR UNDERSTANDING

68. Why is it useful to integrate variants (unions) with records (structs)? Why not leave them as separate mechanisms, as they are in Algol 68 and C?

69. Discuss the type safety problems that arise with variant records. How can these problems be addressed?

70. What is a *tag* (*discriminant*)? How does it differ from an ordinary field?

71. Summarize the rules that prevent access to inappropriate fields of a variant record in Ada.

72. Why might one wish to *constrain* a variable, so that it can hold only one variant of a type?

73. Explain how classes and inheritance can be used to obtain the effect of constrained variant records.

7 Data Types

7.7.2 Dangling References

Memory access errors—dangling references, memory leaks, out-of-bounds access to arrays—are among the most serious 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 15.2.3) 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 proven 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 7.134
Dangling reference detection with tombstones

Tombstones [Lom75, Lom85] allow a language implementation can 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 ◎7.17). 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

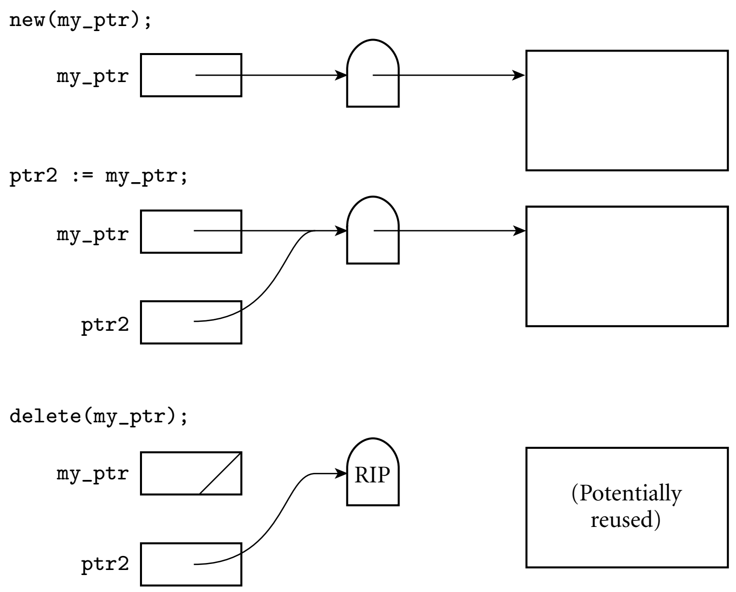


Figure 7.17 **Tombstones.** A valid pointer refers to a tombstone that in turn refers to an object. A dangling reference refers to an "expired" tombstone.

be able to find all tombstones associated with objects in the current stack frame 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 will consider in Section 7.7.3, 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.

Locks and Keys

EXAMPLE 7.135
Dangling reference detection with locks and keys

Locks and *keys* [FL80] are an alternative to tombstones. Their disadvantages are that they work only for objects in the heap, and 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 ◎7.18). 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. ■

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 provably nonredundant access). 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 minimize time and space overhead, most compilers do not by default generate code to check for dangling references. Most Pascal compilers allow the programmer to request dynamic checks, which are usually implemented with locks and keys. In most implementations of C, even optional checks are unavailable.

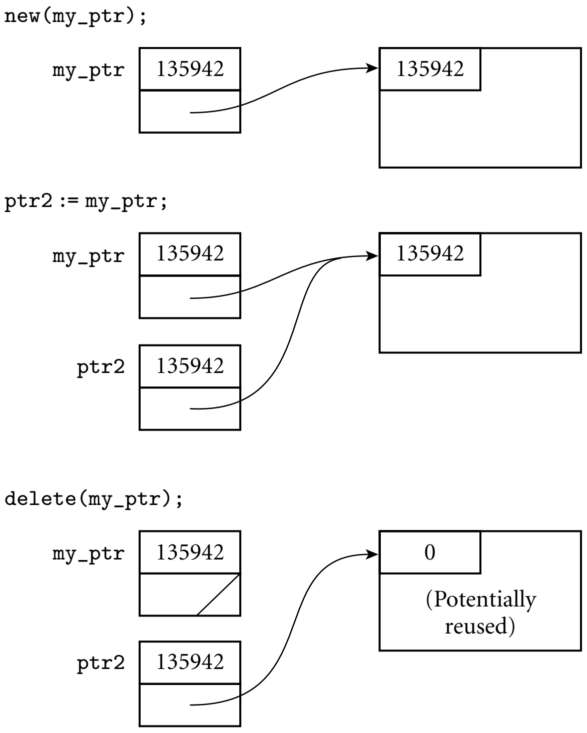


Figure 7.18 **Locks and keys.** A valid pointer contains a key that matches the lock on an object in the heap. A dangling reference is unlikely to match.

CHECK YOUR UNDERSTANDING

74. What are *tombstones*? What changes do they require in the code to allocate and deallocate memory, and to assign and dereference pointers?
75. Explain how tombstones can be used to support *compaction*.
76. What are *locks* and *keys*? What changes do they require in the code to allocate and deallocate memory, and to assign and dereference pointers?
77. Explain why the protection afforded by locks and keys is only probabilistic.
78. Discuss the comparative advantages of tombstones and locks and keys as a means of catching dangling references.

7 Data Types

7.9 Files and Input/Output

The first two subsections below are devoted to interactive and file-based I/O, respectively. Section ©7.9.3 then considers the common special case of text files.

7.9.1 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 8.7, and synchronization in Section 12.3.

A few programming languages—notably Smalltalk and Java—attempt to incorporate a standard set of GUI mechanisms into the language. 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. In a similar vein, Java's original GUI facilities (the Abstract Window Toolkit—AWT) had something of a "least common denominator" look to them.

Smalltalk's GUI is a fundamental part of the language; Java's takes the form of a standard set of library routines. The Java routines and their interface have evolved significantly over time; the current "Swing" libraries have a "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 10.4 and 10.7.

7.9.2 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. These examples, however, are more the exception than the rule. For the most part, data that need to outlive a particular program invocation need 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 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 7.136
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 they are in Pascal:

```
var my_file : file of foo;
```

■

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:

EXAMPLE 7.137
The `open` operation

```
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 7.138
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 magnetic or optical disks, in which the current position can be changed with relative ease. (Depending on technology, modern disks take anywhere from 5 to 200 ms to seek to a new location. Tape drives, by contrast, can take more than a minute. Note that 5 ms is still a very long time—ten million cycles on a 2 GHz processor—so seeking should never be taken casually, even on a disk.) A few languages—notably Pascal—provide no random-access files, though individual implementations may support random access as a nonstandard language extension.

7.9.3 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, each line of a text file ends with a *newline* ("control-J") character, ASCII value 10. On the Macintosh, each line ends with a *carriage return* ("control-M") character, ASCII value 13. In MS-DOS and Windows, each line ends with a carriage return/line feed 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 number) are influenced by the hardware, but most are dictated by the needs of the application and the preferences of the programmer.

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, and RTF can be used to enhance portability.

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 7.139
Formatted output in Fortran

In Fortran, we could write a character string, an integer, and an array of ten 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 7.140
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 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 7.141
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 8.3.3.

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 default and keyword parameters (Section 8.3.3), but Fortran 77 does not, and even in Fortran 90 there is no way to define a subroutine with an *arbitrary* number of parameters.

In Pascal, as in Fortran 77, the parameters of every subroutine are fixed in number and in type. Pascal's read, readln, write, and writeln "routines" are therefore built into the language; they are not part of a library. Each takes a variable number of arguments, the first of which may optionally specify a particular file. Unfortunately, Pascal's formatting mechanisms are much less flexible than those of Fortran; programmers are often forced to implement formatting by hand, using read and write for input and output of individual characters only. In the design of Modula-2, Niklaus Wirth chose to move I/O out of the language proper, and to embed it in a standard library. The designers of Ada took a similar approach. The Modula-2 I/O libraries are relatively primitive: only a modest improvement over character-by-character I/O in Pascal. The Ada libraries are much more extensive, and make heavy use of overloading and default 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 7.142
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 7.143
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 7.144
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 7.145
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 7.146
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 the sidebar on page 353.) The three-character `%lf` placeholder informs the library routine that the corresponding argument is a `double`; the two-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.

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 8.3.3). 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 a new 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.

2 C's `double`s are double-precision IEEE floating-point numbers in most implementations; `float`s 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.

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 character string in C++, one writes

EXAMPLE 7.147
Formatted output in C++

```
my_stream << s;
```

To output a string and an integer one can write

```
my_stream << s << n;
```

This code requires that my_stream be an instance of the ostream (output stream) class defined in the iostream library. The << operator is syntactic sugar for the "operator function" ostream::operator<<, as described in Section 3.5.2. Because << associates left-to-right, the statement above is equivalent to

```
(my_stream.operator<<(s)).operator<<(n);
```

The code works because ostream::operator<< returns a reference to its first argument as its result (as we shall see in Section 8.3.1, C++ supports both a value model and a reference model for variables). ■

EXAMPLE 7.148
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 9.3, 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 7.149
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 7.150
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];
```

The setw manipulator affects the output width of only a single item. 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

79. 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.)
80. What are the comparative advantages of *text* and *binary* files?
81. Describe the end-of-line conventions of Unix, Windows, and Macintosh files.
82. What are the advantages and disadvantages of building I/O into a programming language, as opposed to providing it through library routines?
83. Summarize the different approaches to text I/O adopted by Fortran, Ada, C, and C++.
84. Describe some of the weaknesses of C's `scanf` mechanism.
85. What are *stream manipulators*? How are they used in C++?

7 Data Types

7.12 Exercises

7.27 How would you implement the final version of our `element` type in ML? How would you extract the fields of the variant part? Specifically, suppose you have declared a record to represent copper. How would you specify the equivalent of `copper.source`?

7.28 In Example 6.67 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 ©7.113. Write versions of your iterator for preorder, inorder, and postorder traversals.

7.29 Rewrite the left half of Example ©7.121 in C++ using *references* (see Section 8.3.1).

7.30 Show how variant records in Pascal or unions in C can be used 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.

7.31 Are variant records a form of polymorphism? Why or why not?

7.32 Pascal does not permit the tag field of a variant record to be passed to a subroutine by reference (i.e., as a `var` parameter). Why not?

7.33 Explain how to implement dynamic semantic checks to catch references to uninitialized fields of a tagged variant record in Pascal. 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?

7.34 Explain how to implement dynamic semantic checks to catch references to uninitialized fields of an *untagged* variant record in Pascal. 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?

7.35 We noted in Section ◎7.3.4 that Pascal and Ada require the variant portions of a record to occur at the end, to save space when a particular record is constrained to have a comparatively small variant part. Could a compiler rearrange fields to achieve the same effect, without the restriction on the declaration order of fields? Why or why not?

7.36 In Example 7.88 we noted that reference counts can be used to reclaim tombstones, failing only when the programmer neglects to manually delete the object to which a tombstone refers. Explain how to leverage this observation to catch memory leaks at run time. Does your solution work in all cases? Explain.

7.37 Learn about the *smart pointer* design pattern in C++. Create a smart pointer package that uses tombstones to catch dangling references, and reference counts to reclaim garbage tombstones. Augment your package, as suggested in the preceding exercise, to catch (most) memory leaks.

7.38 Rewrite Example ◎7.146 using `fgets`, `strtol`, `strtod`, etc. (read the `man` pages), so that it is guaranteed not to result in buffer overflow.

7.39 The `readln` and `writeln` procedures of Pascal give special treatment to ends of lines. By contrast, C's `printf` and `scanf` do not; they treat newlines and carriage returns like any other character. What are the comparative advantages of these approaches? Which do you prefer? Why?

7 Data Types

7.13 Explorations

7.50 ML's "grandchild," Haskell, is the focus of much of the current research on functional programming. Learn about its type system. How does it differ from that of ML? What has been added? What has been deleted? What is the rationale for these changes?

7.51 Repeat the previous exercise for Microsoft's F#.

7.52 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 ◎7.9.3.

7.53 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?

8 Subroutines and Control Abstraction

8.2.1 Displays

EXAMPLE 8.63 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 ©8.9 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

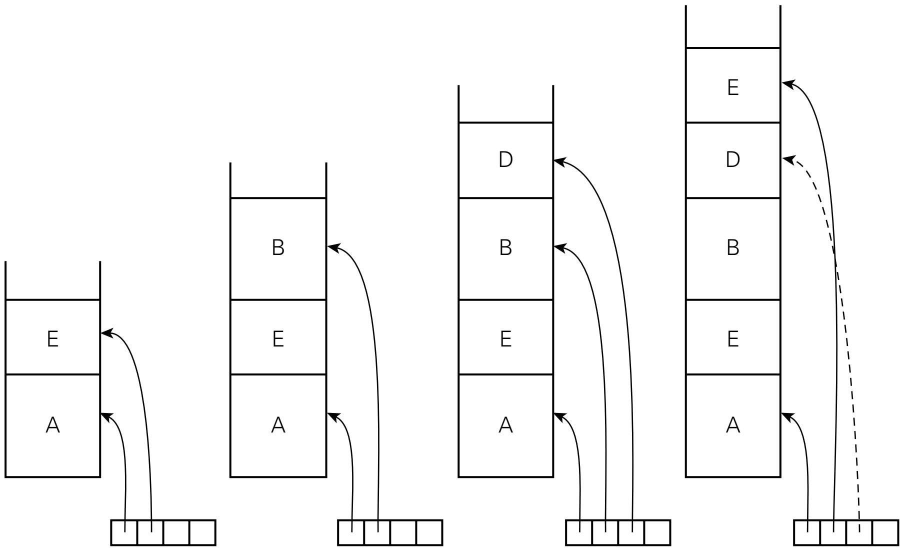


Figure 8.9 **Nonlocal access using a display.** The stack configurations, from left to right, illustrate the contents of the display (at bottom) for a sequence of subroutine calls, assuming the lexical nesting of Figure 8.1. Display elements beyond that of the currently executing subroutine are not used.

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 entry at level j is different from the callee's, so the callee must save it before

DESIGN & IMPLEMENTATION

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.

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 ©8.35), 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 16) 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 9) 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

50. Describe how we access an object at lexical nesting level k in a language implementation based on displays.
51. Why isn't the display typically kept in registers?
52. Explain how to maintain the display during subroutine calls.
53. What special concerns arise when creating closures in a language implementation that uses displays?
54. 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.

8 Subroutines and Control Abstraction

8.2.2 Case Studies: C on the MIPS; Pascal on the x86

To make stack management a bit more concrete, we present a pair of case studies, one for a simple language (C) on a simple RISC machine (the MIPS), the other for a language with nested subroutines (Pascal) on a CISC machine (the x86).

SGI C on the MIPS

An overview of the MIPS architecture can be found in Section ©5.4.5. As noted in that section, register `r31` (also known as `ra`) is special-cased by the hardware to receive the return address in subroutine call (`jal`—jump-and-link) instructions. In addition, register `r29` (also known as `sp`) is reserved by convention for use as the stack pointer, and register `r30` (also known as `fp`) is reserved by convention for the frame pointer, if any. The details presented here correspond to version 7.3.1.3m of the SGI MIPSpro C compiler, generating 64-bit code at optimization level `-O2`. The conventions for 32-bit code are different, and future versions of the compiler may be different as well.

EXAMPLE 8.64
SGI MIPSpro C calling sequence

A typical MIPSpro stack frame appears in Figure ©8.10. The `sp` points to the *last used* location in the stack (note that many other compilers, including some for the MIPS, point the `sp` at the *first unused* location). Since the size of every object in the stack is known at compile time in C, a separate frame pointer is not strictly needed, and the MIPSpro compiler usually does without: it uses displacement-mode offsets from the `sp` for everything in the current stack frame. The principal exception occurs in subroutines whose arguments or local variables are so large that they exceed the reach of displacement addressing; for these the compiler makes use of the `fp`.

Argument Passing Conventions Arguments in the process of being passed to the next routine are assembled at the top of the frame, and are always accessed via offsets from the `sp`. The first eight arguments are passed in integer registers `r4`–`r11` or floating-point registers `f12`–`f19`, depending on type. Additional

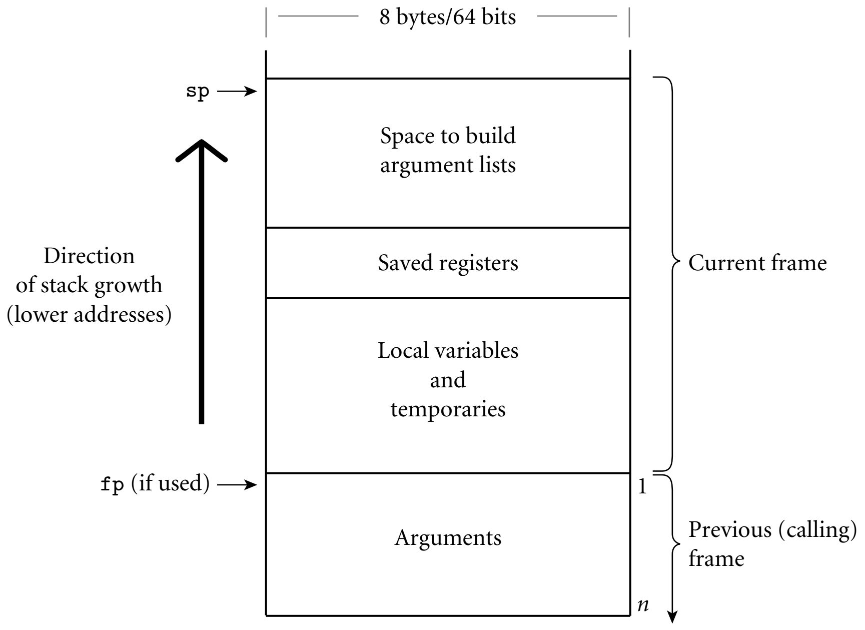

8 bytes/64 bits
sp
Space to build argument lists
Saved registers
Local variables and temporaries
Arguments
Direction of stack growth (lower addresses)
fp (if used)
Current frame
Previous (calling) frame
1
n

Figure 8.10 Layout of the subroutine call stack for the SGI MIPSpro C compiler, running in 64-bit mode. As in Figure 8.2, lower addresses are toward the top of the page.

arguments are passed on the stack. Record arguments (`structs`) are implicitly divided into 64-bit "chunks," each of which is passed as if it were an integer. A large `struct` may be passed partly in registers and partly on the stack.

As noted in the main text, space is reserved in the stack for *all* arguments, whether passed in registers or not. In effect, each subroutine begins with some of its arguments already loaded into registers, and with "stale" values in memory. This is a normal state of affairs; optimizing compilers keep values in registers whenever possible. They "spill" values to memory when they run out of registers, or when there is a chance that the value in memory may be accessed directly (e.g., through a pointer, a reference parameter, or the actions of a nested subroutine). The `fp`, if present, points at the first (top-most) argument.

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 means that arguments need not be "pushed" in the usual sense of the word: the `sp` does not change when they are placed into the stack.

For languages with nested subroutines (C of course is not among them), MIPS compilers generally use register `r2` to pass the static link. In all languages, registers `r2` and `f0` (depending on type) are used to return scalar values from functions. Values of type `long double` are returned in the register pair ⟨`f0, f2`⟩. Record values (`structs`) that will fit in 128 bits are returned in ⟨`r2, r3`⟩. For larger `structs`, the compiler passes a hidden first argument (in `r4`) whose value is the address into which the return value should be placed. 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: at this point they 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.

Calling Sequence Details The calling sequence to maintain the MIPSpro 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 eight scalar arguments (or "chunks" of structs) into registers
3. puts the remaining arguments into the argument build area at the top of the current frame
4. performs a `jal` instruction, which puts the return address in register `ra` and jumps to the target address[1]

The caller-saves registers consist of `r2`–`r15`, `r24`, `r25`, and `f0`–`f23`. In a language with nested subroutines, the caller would place the static link into register `r2` immediately before performing the `jal`.

In its prologue, the callee

1. subtracts the frame size (the distance between the first argument and the `sp` in Figure ©8.10) from the `sp`
2. if the frame pointer is to be used, copies its value into an available temporary register (typically `r2`), then adds the frame size to the `sp`, placing the result in the `fp` (this effectively moves the old `sp` into the `fp`; note that an `add` is as fast as a simple move, so there was no harm in updating the `sp` first)
3. saves any necessary registers into the middle of the newly allocated frame, using the `sp` or, if available, the `fp` as the base for displacement-mode addressing

Saved registers include (a) any callee-saves temporaries (`r16`–`r23` and `f24`–`f31`) whose values may be changed before returning; (b) the `ra`, if the current routine

1 Like all branch instructions on the MIPS, `jal` has an architecturally visible branch delay slot. The load delay slot was eliminated in the MIPS II version of the ISA; all recent MIPS processors are fully interlocked.

is not a leaf or if it uses the ra as an additional temporary; and (c) the temporary register containing the old fp from Step 2, if the current routine needs a frame pointer, or the fp itself if the current routine does not need a frame pointer, but uses the fp as an additional temporary.

In its epilogue, immediately before returning, the callee

1. places the function return value (if any) into r2, r3, f0, f2, or memory as appropriate
2. restores saved registers (if any), using the sp or, if available, the fp as the base for displacement-mode addressing; if the current routine needed a frame pointer, the saved fp is "restored" into a temporary register
3. deallocates the frame by moving the fp into the sp or adding the frame size to the sp
4. moves the value in the temporary register of step 2 (if any), into the fp
5. performs a jr ra instruction (jump to address in register ra)

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 such as gdb and dbx, the compiler generates a variety of assembler pseudo-ops that place information into the object file symbol table. For each subroutine, this information includes the starting and ending addresses of the routine, the size of the stack frame, an indication as to which register (usually sp or fp) is the base for local objects, an indication as to which register (usually ra, if any) holds the return address, and a list of which registers were saved. ■

GNU Pascal on the x86

To illustrate the differences between CISC and RISC machines, our second case study considers the x86, still the world's most popular instruction set architecture. (An overview of the processor appears in Section 5.4.5). To illustrate the handling of nested subroutines and closures, we consider a Pascal compiler, namely version 3.2.2 of the GNU Pascal compiler, gpc. (Ada compilers [e.g., GNU's gnat] handle these features in similar ways, but Ada's many extra features would make the case study much more complex.)

On modern implementations of the x86, ordinary store instructions may make better use of the pipeline than is possible with push. Most modern compilers for the x86, including gcc (on which gpc is based), therefore employ an argument build area similar to that of the previous case study. By default gpc and gcc still use a separate frame pointer, partly for the sake of uniformity with other architectures and languages (gcc is highly portable), and partly to simplify the implementation of library mechanisms that allocate space dynamically in the current stack frame (see Exercise 8.37).

The special instructions for subroutine calls vary significantly from one CISC machine to another. The ones most often used on the x86 today are relatively simple. The call instruction pushes the return address onto the stack, updating

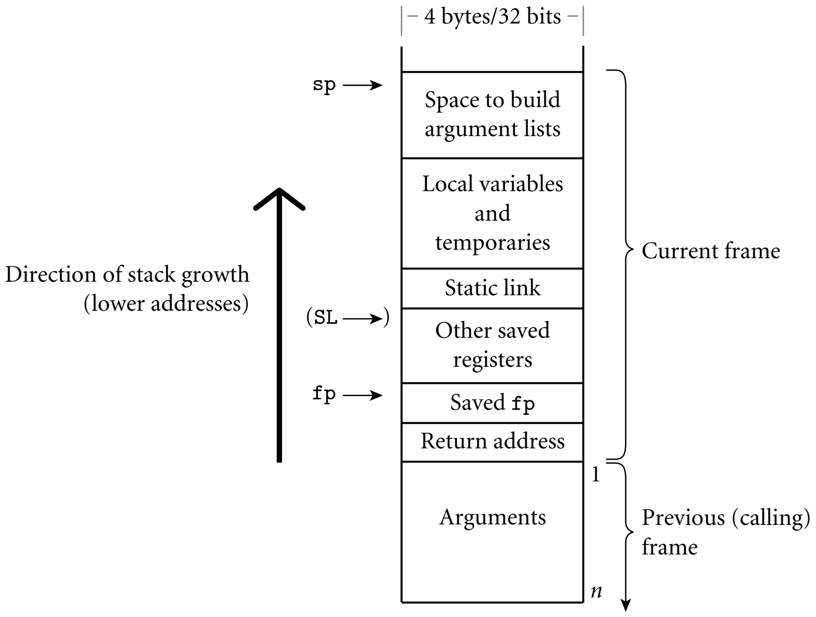


Figure 8.11 Layout of the subroutine call stack for the GNU Pascal compiler, `gpc`. The return address and saved `fp` are present in all frames. All other parts of the frame are optional; they are present only if required by the current subroutine. In x86 terminology, the `sp` is named `esp`; the `fp` is `ebp` (extended base pointer). `SL` marks the location that will be referenced by the static link of any subroutine nested immediately inside this one.

the `sp`, and branches to the called routine. The `ret` instruction pops the return address off the stack, again updating the `sp`, and branches back to the caller. Several additional 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 8.65
Gnu Pascal x86 calling sequence

Argument Passing Conventions Figure 8.11 shows a stack frame for the x86. As in the previous 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`. All arguments are passed in the stack. Register `ecx` is used to pass the static link. That link will point at the last saved register (the saved `fp` if there are no others) in the frame of the lexically surrounding routine, immediately below that routine's own static link, if any.

Functions return integer or pointer values in register `eax`. Floating-point values are returned in the first of the floating-point registers, `st(0)`. For functions that return values of constructed types (records, arrays, or sets), the compiler passes a hidden first argument (on the stack) whose value is the address into which the return value should be placed.

Calling Sequence Details The calling sequence to maintain the `gpc` 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 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 subroutine is declared at the outermost level of lexical nesting. The `call` instruction pushes the return address and jumps to the subroutine.

In its prologue, the callee

1. pushes the `fp` onto the stack, implicitly decrementing the `sp` by 4 (one word)
2. copies the `sp` into the `fp`, establishing the 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 this is not a leaf
5. subtracts the remainder of the frame size from the `sp`

The callee-saves registers are `ebx`, `esi`, and `edi`. 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 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 `push`es in Steps 3 and 4 may be moved after Step 5 and replaced with `fp`-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`, deallocating the frame
4. pops the `fp` off the stack
5. returns

Finally, 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 8.66
Subroutine closure trampoline

Because Pascal allows subroutines to nest, 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 `gpc`, 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 `call`s it. One advantage of this mechanism is its interoperability with `gcc`, in which C functions passed as parameters are simply code addresses. In fact, if *S* is declared at the outermost level of lexical nesting, then `gpc` too can pass an ordinary code address; no trampoline is required. ■

CHECK YOUR UNDERSTANDING

55. For one or both of our 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.
56. On the MIPS some compilers make the `sp` point at the last used word on the stack, while others make it point at the first unused word. On the x86 *all* compilers make it point at the last used word. Why the difference?
57. Why don't the MIPSpro compiler and `gpc` restore caller-saves registers immediately after a call?
58. 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

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 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. Second, many computer security breaches involve so-called *buffer overflow attacks*, in which an intruder exploits the lack of array bounds checking to write code into the stack, where it will be executed when the current subroutine returns. Such attacks are only possible on machines in which writable data are also executable.

8 Subroutines and Control Abstraction

8.2.3 Register Windows

EXAMPLE 8.67
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 8.12). 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 on typical programs.

Register windows have been used in several RISC processors, but only one of these, the SPARC, is commercially significant today. The more recent Intel IA-64 (Itanium) also uses register windows, though it is not a RISC machine. The advantage of windows, of course, is that they reduce the number of loads and stores required for the typical subroutine call. At the same time, register

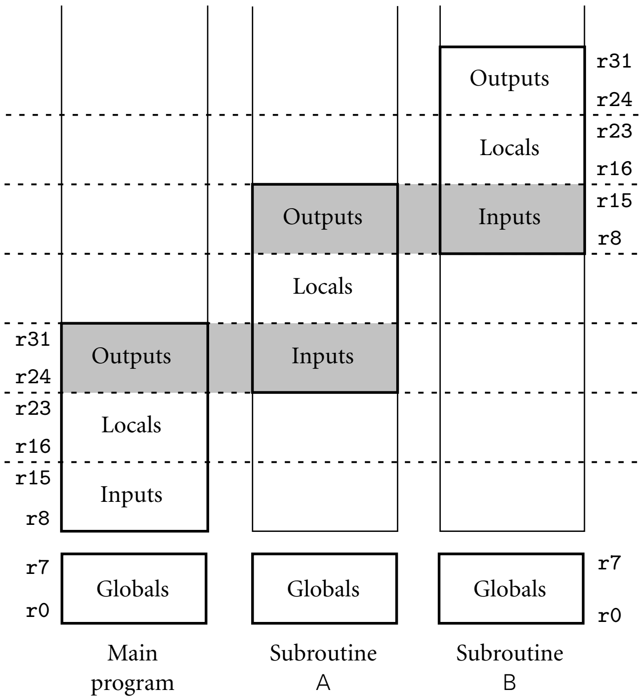


Figure 8.12 Register windows. When the main program calls subroutine A, and again when A calls B, register names `r0`–`r7` continue to refer to the same locations, but register names `r8`–`r31` are changed to refer to a new, overlapping window. High-numbered registers in the caller share locations with low-numbered registers in the callee.

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 8.6), 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

59. What are *register windows*? What purpose do they serve?
60. Which commercial instruction sets include register windows?
61. Explain the concepts of register window *overflow* and *underflow*.
62. Why are register windows a potential problem for multithreaded programs?

8 Subroutines and Control Abstraction

8.3.2 Call by Name

Call by name implements the normal-order argument evaluation described in Section 6.6.2. A call-by-name parameter is re-evaluated 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 is the default in Algol 60; call by value is available as an alternative. In Simula call by value is the default; call by name is the alternative.

To implement call by name, Algol 60 implementations pass a hidden subroutine that evaluates the actual parameter in the caller's referencing environment. The hidden routine is usually called a *thunk*.[2] 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 8.68
Jensen's device

2 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 6.84 (page 276) and the `promise` constructor of Exercise 10.11.

```
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

$$y = \sum_{1 \le x \le 10} 3x^2 - 5x + 2$$

we can simply say

```
y := sum(3*x*x - 5*x + 2, x, 1, 10);
```

■

Label Parameters

Both Algol 60 and Algol 68 allow a label to be passed as a parameter. If a called routine performs a `goto` to such a label, control will usually need to escape the local context, unwinding the subroutine call stack. The unwinding operation depends on the location of the label. For each intervening scope, the `goto` must restore saved registers, deallocate the stack frame, and perform any other operations

DESIGN & IMPLEMENTATION

Call by name

In practice, most uses of call by name in Algol 60 and Simula programs serve simply to allow a subroutine to change the value of an actual parameter; neither language offers 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 Section 3.23 [page 145].) 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.

normally handled by epilogue code. To implement label parameters, Algol implementations typically pass a thunk that performs the appropriate operations for the given label. Note that the target of the label must generally lie in some surrounding scope, where it was visible to the caller under static scoping rules.

Label parameters are 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, a routine that encounters a problem (e.g., invalid input) can perform a `goto` to a label parameter, on the assumption that the label refers to code that performs some remedial operation, or prints an appropriate error message. In more recent languages, label parameters have been replaced by more structured exception handling mechanisms, discussed in Section 8.5.

✓ CHECK YOUR UNDERSTANDING

63. What is *call by name*? What language first provided it? Why isn't it used by the language's descendants?
64. What is *call by need*? How does it differ from call by name?
65. How does a subroutine with call-by-name parameters differ from a macro?
66. What is a *thunk*? What is it used for?
67. What is *Jensen's device*?

DESIGN & IMPLEMENTATION

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.

8 Subroutines and Control Abstraction

8.4.4 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 8.69
Generic `arbiter` class in C++

Figure ©8.13 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 this is not enforced. 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<class T>
class chooser {
public:
    virtual bool operator()(const T& a, const T& b) = 0;
};

template<class T, class C>
class arbiter {
    T* best_so_far;
    C comp;
public:
    arbiter() { best_so_far = 0; }
    void consider(T* t) {
        if (!best_so_far || comp(*t, *best_so_far)) best_so_far = t;
    }
    T* best() {
        return best_so_far;
    }
};
```

Figure 8.13 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 when 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 declarations. 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 [AG90]. 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 8.4.3, 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 ◎7.2.4. It can, as noted in the sidebar on page ◎169, be cast as *unification.*

Unfortunately, per-use instantiation of templates has two significant drawbacks. First, it tends to result in inscrutable error messages. If we define

EXAMPLE 8.70
Instantiation-time errors in C++ templates

```
class foo {
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 65 of source
```

one might hope to receive an error message along the lines of "line 65: foo's operator() method needs to take two arguments of type string&." Instead the Gnu C++ compiler responds

```
simple_best.cc: In member function 'void arbiter<T, C>::consider(T*)
   [with T = std::string, C = foo]':
simple_best.cc:65:   instantiated from here
simple_best.cc:21: error: no match for call to '(foo)
   (std::basic_string<char, std::char_traits<char>,
   std::allocator<char> >&, std::basic_string<char,
   std::char_traits<char>, std::allocator<char> >&)'
```

(Line 21 is the body of method `consider` in Figure ◎8.13.)

Sun's C++ compiler is equally unhelpful:

```
"simple_best.cc", line 21: Error: Cannot cast from std::basic_string<char,
   std::char_traits<char>, std::allocator<char>> to const int.
"simple_best.cc", line 65:   Where: While instantiating
   "arbiter<std::basic_string<char, std::char_traits<char>,
   std::allocator<char>>, foo>::consider(std::basic_string<char,
   std::char_traits<char>, std::allocator<char>>*)".
"simple_best.cc", line 65:   Where: Instantiated from non-template code.
```

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. ■

A second drawback of per-use instantiation is a tendency toward "code bloat": it can be difficult, in the presence of separate compilation, to recognize that the same template has been instantiated with the same arguments in separate compilation units. A program compiled in 20 pieces may have 20 copies of a popular template instance.

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, however, a *cast* would be needed to reassert the original type. No danger was involved, because objects in Java are self-descriptive, and casts 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 casts, 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 casts incur 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 8.71
Generic `arbiter` class in Java

Figure ◎8.14 contains a Java 5 version of our `arbiter` class. It differs from Figure ◎8.13 in several important ways. First, Java insists that all instances of a generic be able to share the same code. Among other things, this means that the `Chooser` to be used by a given 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.) Second, Java requires that the code for the `arbiter` class be manifestly type-safe, independent of any particular instantiation. We must therefore declare `comp` to be a `Chooser`, so we know that it provides a `better` method. This raises the question: what sort of `Chooser` do we need? That is, what should be the generic parameter in the declaration of `comp` (and of the parameter `c` in the `Arbiter` constructor)?

The most obvious choice (*not* the one adopted in Figure ◎8.14) would be `Chooser<T>`. This would allow us to write

```
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 8.14 Generic arbiter in Java.

```
class CaseSensitive implements Chooser<String> {
    public boolean better(String a, String b) {
        return a.compareTo(b) < 1;
    }
}
...
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 8.72
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());
```

```
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 8.15 **Arbiter in Java after type erasure.** No casts are required in this portion of the code (but see the main text for uses).

the compiler would complain. The fix (as shown in Figure ◎8.14) 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`.

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 8.37 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 we need 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 used in Figure ◎8.14, 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

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 casts back to concrete types wherever objects are returned from generic methods. The erased equivalent of Figure ◎8.14 appears in Figure ◎8.15. No casts are required in this portion of the code. On any use of `best`, however, the compiler would insert an implicit cast. The statement

EXAMPLE 8.73
Type erasure and implicit casts

```
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 ◎8.71) 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 version of the code is that the programmer doesn't have to write the casts. 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 8.74
`Unchecked` warnings in Java 5

```
Arbiter<String> csNames = new Arbiter<String>(new CaseSensitive());
Arbiter alias = csNames;            // nongeneric
alias.consider(new Integer(3));     // unsafe
```

DESIGN & IMPLEMENTATION

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 byte code 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 second line of this example, because we have invoked method `consider` on a "raw" (nongeneric) `Arbiter` without explicitly casting 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 8.75
Java 5 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 casts 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 8.76
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 8.77
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

```
public delegate bool Chooser<T>(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(t, bestSoFar)) bestSoFar = t;
        initialized = true;
    }
    public T Best() {
        return bestSoFar;
    }
}
```

Figure 8.16 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 8.78
Generic `arbiter` class in C#

A C# version of our `Arbiter` class appears in Figure ◎8.16. One small difference with respect to Figure ◎8.14 appears 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.

More interesting differences from Figure ◎8.14 are the definition of `Chooser` as a *delegate*, rather than an interface, and the lack of lower bounds (uses of the `super` keyword) in parameter and field declarations. These issues are connected. C# allows us to specify an *upper* bound as a type constraint; we did so in the `sort` routine of Example 8.38. There is no direct equivalent, however, for Java's lower bounds. We can work around the problem in the `Arbiter` example by exploiting the fact that `Chooser` has only one method (named `better` in Figure ◎8.14).

As described in Section 3.6.3, a C# delegate is a first-class subroutine. The delegate declaration in Figure ◎8.16 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

```
public static bool CaseSensitive(String a, String b) {
    return String.CompareOrdinal(a, b) < 1;
    // use Unicode order, in which upper-case letters come first
}
public 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

68. Why is it difficult to produce high-quality error messages for misuses of C++ templates?
69. What is *template metaprogramming*?
70. Explain the difference between *upper bounds* and *lower bounds* in Java type constraints. Which of these does C# support?
71. What is *type erasure*? Why is it used in Java?
72. Under what circumstances will a Java compiler issue an "unchecked" generic warning?
73. For what two main reasons are C# generics less susceptible to "code bloat" than C++ templates are?

74. Why must fields of generic parameter type be explicitly initialized in C#?

75. For what two main reasons are C# generics often more efficient than comparable code in Java?

76. How does a C# *delegate* differ from an interface with a single method (e.g., the C++ `chooser` of Figure ◎8.13? How does it differ from a function pointer in C?

8 Subroutines and Control Abstraction

8.6.3 Implementation of Iterators

EXAMPLE 8.79
Coroutine-based iterator invocation

Consider the following `for` loop from Example 6.63 (page 263):

```
for i in range(first, last, step):
    ...
```

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 8.80
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 (page 264). ■

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 (at the top 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 ©8.45).

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 8.81
Iterator usage in C#

```
for (IEnumerator i = myTree.GetEnumerator(); i.MoveNext();) {
    object o = i.Current;
    Console.WriteLine(o.ToString());
}
```
■

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:

EXAMPLE 8.82
Implementation of C# iterators

```
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 ©8.17. 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 ©8.46. 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

77. Describe the "obvious" implementation of iterators using coroutines.
78. Explain how the state of multiple active iterators can be maintained in a single stack.
79. 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 8.17 Iterator object equivalent of a true iterator in C#. This handwritten code corresponds to Example 8.82. 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.

8 Subroutines and Control Abstraction

8.6.4 Discrete Event Simulation

EXAMPLE 8.83
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 8.84
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 8.85

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 8.86

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 8.87

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 8.88
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 8.89
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 microprocessors. 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

80. Summarize the computational model of discrete event simulation. Explain the significance of the time-based priority queue.
81. When building a discrete event simulation, how does one decide which things to model with coroutines, and which to model with data structures?
82. Are all inactive coroutines guaranteed to be in the priority queue? Explain.

8 Subroutines and Control Abstraction

8.9 Exercises

8.35 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?

8.36 Elaborate on the reasons why parameters on the MIPS may 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.

8.37 Most versions of the C library include a function, `alloca`, that dynamically allocates space within the current stack frame.[3] It has two advantages over the usual `malloc`, which allocates space in the stack: 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 MIPSpro calling conventions described in Section ©8.2.2?

8.38 Explain how to extend the conventions of Figure ©8.11 and Section ©8.2.2 to accommodate arrays whose bounds are not known until elaboration time (as discussed in Section 7.4.2). What ramifications does this have for the use of separate stack and frame pointers?

3 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.

8.39 In both the MIPSpro and gpc case studies, arguments were placed into the argument build area in "reverse" order, with the first argument at the top. Explain why this is important. (Hint: Consider subroutines with variable numbers of arguments, as discussed in Section 8.3.3.)

8.40 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?

8.41 Explain how you might implement setjmp and longjmp on a SPARC.

8.42 Continuing Example ©8.71, the call

```
csNames.consider(null);
```

will generate a run-time exception, because String.compareTo is not designed to take null arguments.

(a) Modify Figure ©8.14 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 ©8.16 and Example ©8.78. How should you handle lower bounds when you need both Better and Valid?

8.43 **(a)** Modify your solution to Exercise 8.22 so that the comparison routine is an explicit generic parameter, reminiscent of the chooser of Figure ©8.13.

(b) Give an alternative solution in which the comparison routine is an extra parameter to sort.

8.44 Consider the C++ program shown in Figure ©8.18. 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.

8.45 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, three-level (six-node) binary tree.

8.46 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.

8.47 One source of inaccuracy in the traffic simulation of Section ©8.6.4 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.

```
#include <iostream>
#include <list>
using std::cout;
using std::list;

template<class 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 8.18 **Coercion and generics in C++.** The compiler refuses to accept the final call to `first_n`.

8 Subroutines and Control Abstraction

8.10 Explorations

8.58 Research the calling sequence used by SGI compilers when running on the MIPS in 32-bit mode. Compare and contrast to the conventions of Section ©8.2.2. 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.

8.59 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.

8.60 As another 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.

8.61 Investigate the concepts of *covariance* and *contravariance* in object-oriented languages. Explain what they have to do with upper and lower bounds (`? extends T` and `? super T`) on Java type parameters.

9

Data Abstraction and Object Orientation

9.5 Multiple Inheritance

EXAMPLE 9.50
Deriving from two base classes (reprise)

Recall our simple example in C++:

```
class student : public person, public gp_list_node { ...
```

To implement multiple inheritance, we must be able to generate both a "`person` view" and a "`gp_list_node` view" of a `student` object on demand, for example when assigning a reference to a `student` object into a `person` or `gp_list_node` 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 9.51
(Nonrepeated) multiple inheritance

For the other base class (`gp_list_node`) 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 ©9.7. It is based loosely on the implementation described by Ellis and Stroustrup [ES90, Chap. 10]. Because the `gp_list_node` fields of a `student` follow the `person` fields, the assignment of a reference to a `student` object into a variable of type `gp_list_node*` 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 `gp_list_node` defines a virtual method `debug_print` that is supposed to dump a printable representation of the contents of the node to standard output.) Generalization to three or more base classes is straightforward; see Exercise ©9.22.

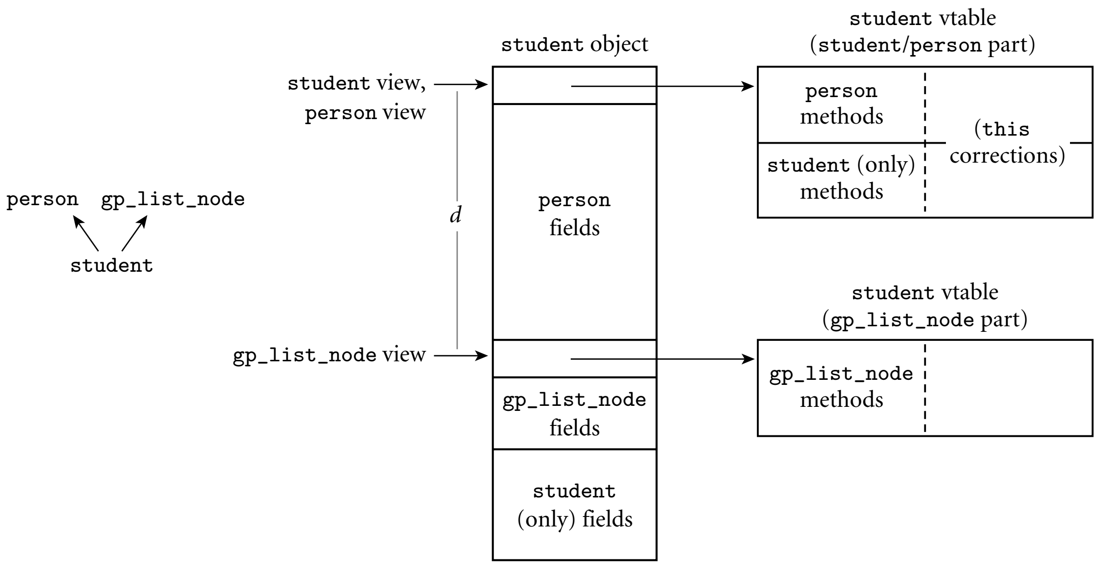


Figure 9.7 **Implementation of (nonrepeated) multiple inheritance.** The size d of the `person` portion of the object is a compile-time constant. We access the `gp_list_node` portion of the vtable by adding d to the address of a `student` object before indirecting. Likewise, we create a `gp_list_node` 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.

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 the `person/student` portion of the vtable is stored in the beginning of the object. The address of the `gp_list_node` portion of the vtable is stored at offset d. Note that both parts of the vtable are specific to class `student`. In particular, the `gp_list_node` part of the vtable is *not* shared by objects of class `gp_list_node`, because the contents of the tables will be different if `student` has overridden any of `gp_list_node`'s virtual methods. ■

EXAMPLE 9.52
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 9.4.3 for single inheritance. To call a virtual method originally defined in `gp_list_node`, we must first add the offset d to our object's address, in order to find the address of the `gp_list_node` portion of the vtable. Then we can index into this `gp_list_node` 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 `debug_print` (even though it probably ought to). If our object is of class `student`, we should pass a `gp_list_node` view of it to `debug_print`: the address of the object, plus d. If, however, our object is of some class (`transfer_student`, perhaps) that does override `debug_print`, then we should pass a `transfer_student` view to `debug_print`. 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 `gp_list_node` fields appear! ■

EXAMPLE 9.53
`This` correction

A common solution is for vtable entries 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 ◎9.7, the "`this` correction" field of the vtable entry for `debug_print` would contain $-d$ if `debug_print` was overridden by `student`, and zero otherwise. In the `gp_list_node` 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 `debug_print` is the third virtual method of `gp_list_node`, then the code to call `my_student.debug_print` would look something like this:

```
r1 := my_student                 -- student view of object
r1 := r1 + d                     -- gp_list_node 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. 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. ■

9.5.1 Semantic Ambiguities

In addition to implementation complexities (only some of which we have discussed so far), multiple inheritance introduces potential semantic problems.

EXAMPLE 9.54
Methods found in more than one base class

Suppose that both `gp_list_node` and `person` define a `debug_print` method. If we have a variable `s` of type `student*` and we call `s->debug_print`, 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. In Eiffel we can use the feature renaming mechanism to get rid of naming conflicts when defining a derived class. In C++ we must redefine the ambiguous member explicitly:

```
void student::debug_print() {
    person::debug_print();
    gp_list_node::debug_print();
}
```

Here we have chosen to call the `debug_print` 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::debug_print_person() {
    person::debug_print();
}
void student::debug_print_list_node() {
    gp_list_node::debug_print();
}
```

■

EXAMPLE 9.55
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 [Str97, Sec. 25.6], we can solve the problem by interposing an "interface" class between each base class and the derived class:

```
class person_interface : public person {
public:
    virtual void debug_print_person() = 0;
    void debug_print() { debug_print_person(); }
        // overrides person::debug_print
};
```

DESIGN & IMPLEMENTATION

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 ©9.28) in which the cost of modifying `this` is paid only when the correction is nonzero.

```
class list_node_interface : public gp_list_node {
public:
    virtual void debug_print_list_node() = 0;
    void debug_print() { debug_print_list_node(); }
        // overrides gp_list_node::debug_print
};
class student : public person_interface, public list_node_interface {
public:
    void debug_print_person() { ...
    void debug_print_list_node() { ...
    ...
};
```

We leave it as an exercise (◎9.23) to show what happens if we assign a student object into a variable p of type person* and then call p->debug_print(). ■

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.

EXAMPLE 9.56
Repeated multiple inheritance

For example, suppose in our administrative computing system that we would like to keep all professors in the same department on a linked list. Like class student, we might want class professor to inherit from both person and gp_list_node:

```
class professor : public person, public gp_list_node { ...
```

Furthermore, suppose that professors occasionally take courses 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 gp_list_node twice, through both 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 gp_list_node—two predecessors and two successors, one set for linking into the list of nonmatriculated students and another for linking into the faculty list for some department:

$$\begin{tikzcd}
\text{gp\_list\_node} & & \text{person} & & \text{gp\_list\_node} \\
& \text{student} \arrow[ul] \arrow[ur] & & \text{professor} \arrow[ul] \arrow[ur] & \\
& & \text{student\_prof} \arrow[ul] \arrow[ur] & &
\end{tikzcd}$$

The gp_list_node 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 9.57
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 gp_list_node { ...
class professor : public virtual person, public gp_list_node { ...
```

In this case the members of class person are shared when inherited over multiple paths, while the members of class gp_list_node are replicated. ■

EXAMPLE 9.58
Replicated inheritance in Eiffel

Replicated inheritance of individual features can be obtained in Eiffel through the renaming mechanism described in Section 9.2.2:

```
class student inherit person; gp_list_node ...
class professor inherit person; gp_list_node ...

class student_prof
inherit
    student
        rename
            prev as prev_student,
            next as next_student
        end;
    professor
        rename
            prev as prev_prof,
            next as next_prof
        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 above explicitly; there is no other renaming mechanism. ■

9.5.2 Replicated Inheritance

EXAMPLE 9.59
Using replicated inheritance

Replicated inheritance introduces no serious implementation problems beyond those of nonrepeated multiple inheritance. As shown in Figure 9.8, 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

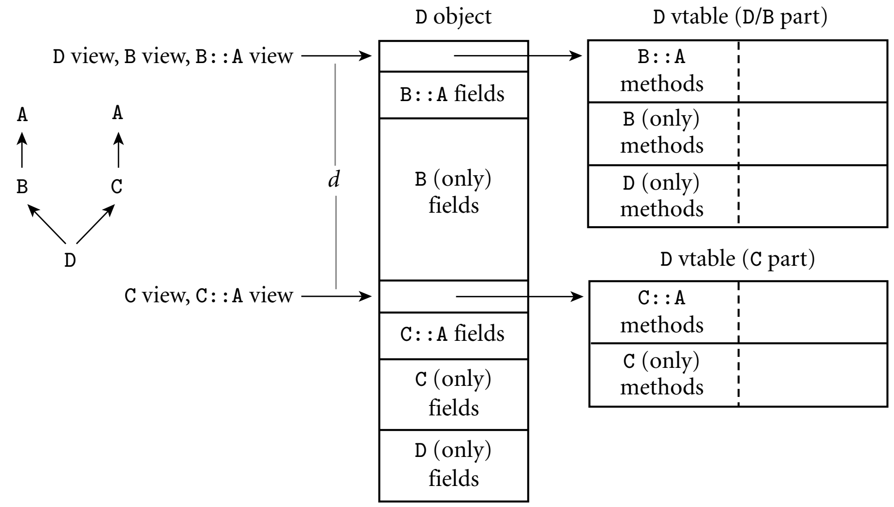


Figure 9.8 **Implementation of replicated multiple inheritance.** Each base class contains a complete copy of class `A`. As in Figure ◎9.7, the vtable for class `D` is split into two parts, one for each base class, and each vtable entry consists of a ⟨method address, `this` correction⟩ pair.

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:

```
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 ◎9.53, vtable entries will need to consist of ⟨method address, `this` correction⟩ pairs. ■

9.5.3 Shared Inheritance

EXAMPLE 9.60
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:

```
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 one it wants. Alternatively, through renaming, it can retain access to all versions. ■

EXAMPLE 9.61
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 9.9, 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 9.53), but involves one more memory

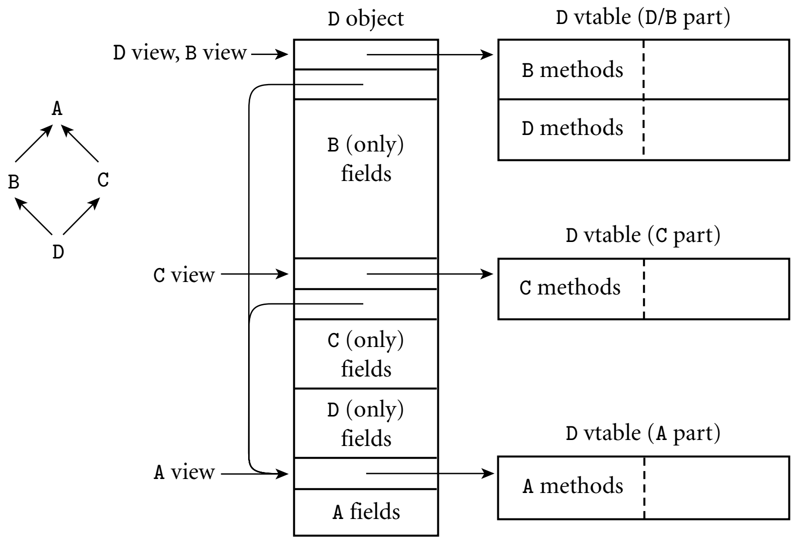


Figure 9.9 **Implementation of shared multiple inheritance.** Objects of class `B`, `C`, and `D` contain the address of their `A` components at a compile-time constant offset (in this case, immediately after the vtable address). As in Figures ◎9.7 and ◎9.8, `this` corrections for virtual methods in vtable entries are relative to the view of the class in which the method was declared (i.e., through which the vtable was accessed).

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 ◎9.9 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 ◎9.7 and ◎9.8. 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 ◎9.29).

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 ◎9.30.

9.5.4 Mix-In Inheritance

Before leaving the topic of multiple inheritance, we return briefly to the notion of a base class composed entirely of abstract methods, as mentioned in passing in

Section 9.4.2. Such a class is usually called an *interface*. It has neither data members nor implementations of its methods. It is therefore immune to most of the semantic ambiguities and implementation complexities of multiple inheritance. Interfaces appear in Java, C#, and Ada 2005.

Inheritance from one "real" base class and an arbitrary number of interfaces is known as *mix-in* inheritance—the virtual methods of the interface are "mixed into" the methods of the derived class. It may be stretching things a bit to speak of "inheriting" an interface, since the derived class must provide a definition for each of the interface's methods. Interfaces do, however, facilitate code reuse through polymorphism. If a formal parameter of a subroutine is declared to have an interface type, then any class that implements (inherits from) that interface can be passed as the corresponding actual parameter. The classes of objects that can legitimately be passed need not have a common class ancestor.

EXAMPLE 9.62
Mixing interfaces into a derived class

As an example, suppose that we have been given general-purpose Java code that will sort objects according to some textual field, display a graphic representation of an object within a web browser window (hiding and refreshing as appropriate), and store references to objects by name in a dictionary data structure. Each of these capabilities would be represented by an interface. If we have already developed some complicated class of objects `widget`, we can make use of the general-purpose code by mixing the appropriate interfaces into classes derived from `widget`, as shown in Figure Ⓒ9.10. ■

EXAMPLE 9.63
Compile-time implementation of mix-in inheritance

As noted in Section 9.4.3, Java implementations usually look methods up by name at run time. In this case, the methods of an interface can simply be added to the method dictionary of any class that implements the interface. To implement mix-in inheritance without run-time method lookup, one simple approach is to augment the representation of objects of the class with the addresses of vtables for the implemented interfaces, as shown in Figure Ⓒ9.11. Additional vtable pointers, like additional data members, are added to the end of the representation of objects of the base class to create the representation of the derived class. If interfaces and data members are added at several levels of the class hierarchy, then vtable pointers and data members may be interspersed at arbitrary offsets within objects. ■

✓ CHECK YOUR UNDERSTANDING

44. Give a few examples of the semantic ambiguities that arise when a class has more than one base class.

45. Explain the distinction between replicated and shared multiple inheritance. When is each desirable?

46. Explain how even nonrepeated multiple inheritance introduces the need for multiple *views* of (the implementation of) an object, and for "`this` correction" fields in vtables.

47. Explain how shared multiple inheritance introduces the need for an additional level of indirection when accessing fields of certain parent classes.

```
public class widget { ... }
interface sortable_object {
    String get_sort_name();
    bool less_than(sortable_object o);
    // All methods of an interface are automatically public.
}
interface graphable_object {
    void display_at(Graphics g, int x, int y);
    // Graphics is a standard library class that provides a context
    // in which to render graphical objects.
}
interface storable_object {
    String get_stored_name();
}
class named_widget extends widget implements sortable_object {
    public String name;
    public String get_sort_name() {return name;}
    public bool less_than(sortable_object o) {
        return (name.compareTo(o.get_sort_name()) < 0);
        // compareTo is a method of the standard library class String.
    }
}
class augmented_widget extends named_widget
        implements graphable_object, storable_object {
    ...         // more data members
    public void display_at(Graphics g, int x, int y) {
        ...     // series of calls to methods of g
    }
    public String get_stored_name() {return name;}
}
...
class sorted_list {
    public void insert(sortable_object o) { ...
    public sortable_object first() { ...
    ...
}
class browser_window extends Frame {
    // Frame is the standard library class for windows.
    public void add_to_window(graphable_object o) { ...
    ...
}
class dictionary {
    public void insert(storable_object o) { ...
    public storable_object lookup(String name) { ...
    ...
}
```

Figure 9.10 **Interface classes in Java.** By implementing the `sortable_object` interface in `named_widget` and the `graphable_object` and `storable_object` interfaces in `augmented_widget`, we obtain the ability to pass objects of those classes to and from such routines as `sorted_list.insert`, `browser_window.add_to_window`, and `dictionary.insert`.

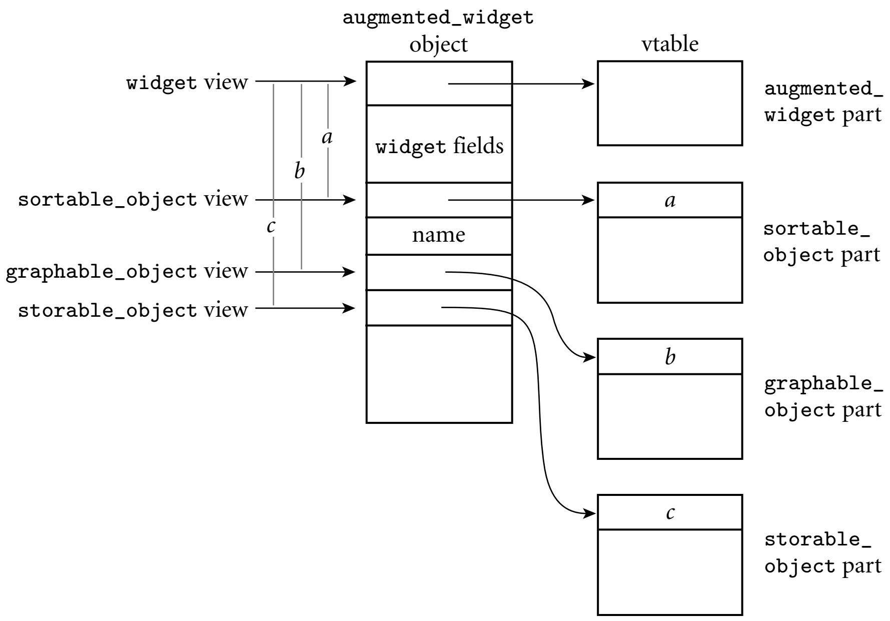


Figure 9.11 **Implementation of mix-in inheritance.** Objects of class `augmented_widget` contain four vtable addresses, one for the class itself (as in Figure 9.3), and three for the implemented interfaces. The view of the object that is passed to interface routines points directly at the relevant vtable pointer. The vtable then begins with a single `this` correction, used by all of its methods to regenerate a pointer to the object itself.

48. What is an *interface*, as defined by Java, C#, or Ada 2005? How is it related to *mix-in* style inheritance?

49. Why is mix-in inheritance simpler to implement than other styles of multiple inheritance?

9 Data Abstraction and Object Orientation

9.6.1 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 9.3.1, 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 9.64
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 9.65
Mixfix messages

As described in Section 6.1, multiargument messages have multiword ("mix-fix") 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 9.66
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 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 9.67
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 $\texttt{n}^{10}$. 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 9.68
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 9.69
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 9.70
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 8.5.3):

```
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.

We end this subsection by observing that recursion works at least as well in Smalltalk as it does in other imperative languages.

EXAMPLE 9.71
Recursion in Smalltalk

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 boldface 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.

9 Data Abstraction and Object Orientation

9.8 Exercises

9.22 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.

9.23 Consider the person_interface and list_node_interface classes described in Example ◎9.55. If student is derived from person_interface and list_node_interface, explain what happens in the following method call:

```
student s;
person *p = &s;
...
p.debug_print();
```

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 ◎9.8, without shared inheritance.

9.24 Given the inheritance tree of Example ◎9.56, show a representation for objects of class student_prof. You may want to consult Figures ◎9.7, ◎9.8, and ◎9.9.

9.25 Given the memory layout of Figure ◎9.7 and the following declarations:

```
student& sr;
gp_list_node& nr;
```

show the code that must be generated for the assignment

```
nr = sr;
```

(Pitfall: Be sure to consider null pointers.)

9.26 Consider the following code, written in a language like Java, with mix-in inheritance:

```
class A {
    private int a;
    ...
}
class B extends A implements Runnable {
    private int b;
    ...
}
...
Runnable r = new B();
```

(a) Draw a diagram of the implementation of a B object, showing fields (data members) and vtable(s).

(b) Show (in pseudo-assembly language) the calling sequence for `r.run()` (caller side only). You may assume that `r` has been loaded into register r1, and that `run` is a method defined by the `Runnable` interface. You may use as many registers as you need. You need not preserve r1. You may assume that the `this` parameter is to be passed in register r3.

9.27 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?

9.28 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 ©9.53? 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.

9.29 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 7.4.2. For further details, consult the compiler text of Wilhelm and Maurer [WM95, Sec. 5.3].)

9.30 In Figure ©9.9, 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?

9.31 In Ruby, an interface (mix-in) can provide not only method signatures (names and parameter lists), but also method *code*. (It can't provide data members; that would be multiple inheritance.) Would this feature make sense in Java? Explain.

9.32 Consider the Smalltalk implementation of Euclid's algorithm, presented at the end of Section ©9.6.1. Trace the messages involved in evaluating `4 gcd: 6`.

9 Data Abstraction and Object Orientation

9.9 Explorations

9.38 Figure out how multiple inheritance is implemented in your local C++ compiler. How closely does it follow the strategy of Sections 9.5.2 and 9.5.3? What rationale do you see for any differences?

9.39 Explore the implementation of mix-in inheritance in a just-in-time (native code) Java compiler. Does it follow the strategy of Section 9.5.4? How efficient is it?

9.40 Learn how multiple inheritance is implemented in Perl and Python (you might begin by reading Section 13.4.4). Describe the differences with respect to Sections 9.5.2 and 9.5.3. Discuss the advantages and drawbacks of dynamic typing in object-oriented languages.

10 Functional Languages

10.6 Theoretical Foundations

EXAMPLE 10.44
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 10.45
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:

$$\text{sqrt} \equiv \{(x, y) \in \mathcal{R} \times \mathcal{R} \mid y > 0 \wedge x = y^2\}$$
$$\text{plus} \equiv \{(x, y, z) \in \mathcal{R} \times \mathcal{R} \times \mathcal{R} \mid z = x + y\}$$

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 10.46
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 10.47
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 10.48
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 inference system (Section ◎7.2.4). ■

EXAMPLE 10.49
Curried functions as sets

Using the notion of "currying" from Section 10.5, 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 ©10.6.3.

10.6.1 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 ©10.21.) 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 10.50 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 10.51 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:

$$\begin{aligned}
\textit{expr} &\longrightarrow \texttt{name} \mid \textit{number} \mid \lambda\ \texttt{name}\ .\ \textit{expr} \mid \textit{func}\ \textit{arg} \\
\textit{func} &\longrightarrow \texttt{name} \mid (\ \lambda\ \texttt{name}\ .\ \textit{expr}\ ) \mid \textit{func}\ \textit{arg} \\
\textit{arg} &\longrightarrow \texttt{name} \mid \textit{number} \mid (\ \lambda\ \texttt{name}\ .\ \textit{expr}\ ) \mid (\ \textit{func}\ \textit{arg}\ )
\end{aligned}$$

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 10.52
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:

$$\lambda x.\text{times}\ x\ x$$

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 10.53
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 $\lambda x.\lambda y.\text{times}\ x\ y$. In the inner expression ($\lambda y.\text{times}\ x\ y$), 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 10.54
Naming functions for future reference

If we wish to refer to them later, we can give expressions names:

$$\begin{aligned}
\text{square} &\equiv \lambda x.\text{times}\ x\ x\\
\text{identity} &\equiv \lambda x.x\\
\text{const7} &\equiv \lambda x.7\\
\text{hypot} &\equiv \lambda x.\lambda y.\text{sqrt}\ (\text{plus}\ (\text{square}\ x)\ (\text{square}\ y))
\end{aligned}$$

Here $\equiv$ is a metasymbol meaning, roughly, "is an abbreviation for." ■

EXAMPLE 10.55
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

$$\lambda x.F\ x \rightarrow_\eta F$$

■

$$\begin{array}{lll}
 & (\underline{\lambda f}.\lambda g.\lambda h.fg(h\,h))\underline{(\lambda x.\lambda y.x)}h(\lambda x.x\,x) & \\
\rightarrow_\beta & (\lambda g.\underline{\lambda h}.(\lambda x.\lambda y.x)g(\underline{h\,h}))h(\lambda x.x\,x) & (1) \\
\rightarrow_\alpha & (\underline{\lambda g}.\lambda k.(\lambda x.\lambda y.x)g(k\,k))\underline{h}(\lambda x.x\,x) & (2) \\
\rightarrow_\beta & (\underline{\lambda k}.(\lambda x.\lambda y.x)h(k\,k))\underline{(\lambda x.x\,x)} & (3) \\
\rightarrow_\beta & (\underline{\lambda x}.\lambda y.x)\underline{h}((\lambda x.x\,x)\ (\lambda x.x\,x)) & (4) \\
\rightarrow_\beta & (\underline{\lambda y}.h)\underline{((\lambda x.x\,x)\ (\lambda x.x\,x))} & (5) \\
\rightarrow_\beta & h & (6)
\end{array}$$

Figure 10.3 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.

EXAMPLE 10.56
Delta reduction for arithmetic

To accommodate arithmetic we will also allow an expression of the form 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:

$$\begin{array}{rcl}
\text{plus}\ 2\ 3 & \rightarrow_\delta & 5 \\
\text{minus}\ 5\ 2 & \rightarrow_\delta & 3 \\
\text{times}\ 2\ 3 & \rightarrow_\delta & 6
\end{array}$$

■

Beta reduction resembles the use of call by name parameters (Section 8.3.1). 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

EXAMPLE 10.57
Eta reduction

$$\lambda x.\text{square}\ x \rightarrow_\eta \text{square}$$

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. ■

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 ©10.3. In line (2) of this derivation we have to

EXAMPLE 10.58
Reduction to simplest form

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 10.4. 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 10.59

Nonterminating applicative-order reduction

The example expression of Figure 10.3 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:

$$\begin{array}{ll}
 & (\underline{\lambda x}.x\,x)\,\underline{(\lambda x.x\,x)} \\
\to_\beta & (\underline{\lambda x}.x\,x)\,\underline{(\lambda x.x\,x)} \\
\to_\beta & (\underline{\lambda x}.x\,x)\,\underline{(\lambda x.x\,x)} \\
\to_\beta & (\underline{\lambda x}.x\,x)\,\underline{(\lambda x.x\,x)} \\
\ldots &
\end{array}$$

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.*

10.6.2 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 10.60
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:

$$\text{if} \equiv \lambda c.\lambda t.\lambda e.c\ t\ e$$

Consider:

$$\begin{aligned} \text{if } T\ 3\ 4 &\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)\ (\lambda x.\lambda y.x)\ 3\ 4 \\ &\to_\beta^* (\lambda x.\lambda y.x)\ 3\ 4 \\ &\to_\beta^* 3 \end{aligned}$$

$$\begin{aligned} \text{if } F\ 3\ 4 &\equiv (\lambda c.\lambda t.\lambda e.c\ t\ e)\ (\lambda x.\lambda y.y)\ 3\ 4 \\ &\to_\beta^* (\lambda x.\lambda y.y)\ 3\ 4 \\ &\to_\beta^* 4 \end{aligned}$$

■

Functions like equal and greater_than can be defined to take numeric values as arguments, returning T or F.

EXAMPLE 10.61
Beta abstraction for recursion

Recursion is a little tricky. An equation like

$$\begin{aligned} \text{gcd} \equiv\ & \lambda a.\lambda b.(\text{if (equal } a\ b)\ a \\ & \quad (\text{if (greater\_than } a\ b)\ (\text{gcd (minus } a\ b)\ b)\ (\text{gcd (Minus } b\ a)\ a))) \end{aligned}$$

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):

$$\begin{aligned} \text{gcd} \equiv\ & (\lambda g.\lambda a.\lambda b.(\text{if (equal } a\ b)\ a \\ & \quad (\text{if (greater\_than } a\ b)\ (g(\text{minus } a\ b)\ b)\ (g(\text{minus } b\ a)\ a))))\ \text{gcd} \end{aligned}$$

Now our equation has the form

$$\text{gcd} \equiv f\ \text{gcd}$$

where f is the perfectly well-defined (nonrecursive) lambda expression

$$\begin{aligned} & \lambda g.\lambda a.\lambda b.(\text{if (equal } a\ b)\ a \\ & \quad (\text{if (greater\_than } a\ b)\ (g\ (\text{minus } a\ b)\ b)\ (g\ (\text{minus } b\ a)\ a))) \end{aligned}$$

Clearly gcd is a fixed point of f. ■

EXAMPLE 10.62
The fixed-point combinator **Y**

As it turns out, for any function f given by a lambda expression, we can find the least fixed point of f, if there is one, 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 ◎10.9). In the case of our gcd function, we have

$$\begin{aligned}\mathsf{gcd} \;\equiv\; & (\lambda h.(\lambda x.h(x\,x))\,(\lambda x.h(x\,x)))\\ & (\lambda g.\lambda a.\lambda b.(\mathsf{if}\ (\mathsf{equal}\ a\ b)\ a\\ & \quad (\mathsf{if}\ (\mathsf{greater\_than}\ a\ b)\ (g(\mathsf{minus}\ a\ b)\ b)\ (g(\mathsf{minus}\ b\ a)\ a))))\end{aligned}$$

Figure ◎10.4 traces the evaluation of gcd 4 2. Given the existence of the **Y** combinator, most authors permit recursive "definitions" of functions, for convenience. ■

10.6.3 Structures

EXAMPLE 10.63
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:

$$\begin{aligned}\mathsf{cons} &\equiv \lambda a.\lambda d.\lambda x.x\ a\ d\\ \mathsf{car} &\equiv \lambda l.l\ \mathsf{select\_first}\\ \mathsf{cdr} &\equiv \lambda l.l\ \mathsf{select\_second}\\ \mathsf{nil} &\equiv \lambda x.T\\ \mathsf{null?} &\equiv \lambda l.l(\lambda x.\lambda y.F)\end{aligned}$$

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 10.64
List operator identities

Using these definitions we can see that

$$\begin{aligned}\mathsf{car}(\mathsf{cons}\ A\ B) &\equiv (\lambda l.l\ \mathsf{select\_first})\ (\mathsf{cons}\ A\ B)\\ &\rightarrow_\beta (\mathsf{cons}\ A\ B)\ \mathsf{select\_first}\\ &\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ \mathsf{select\_first}\\ &\rightarrow_\beta^* (\lambda x.x\ A\ B)\ \mathsf{select\_first}\\ &\rightarrow_\beta \mathsf{select\_first}\ A\ B\\ &\equiv (\lambda x.\lambda y.x)\ A\ B\\ &\rightarrow_\beta^* A\end{aligned}$$

$$\begin{aligned}
\text{gcd}\ 2\,4 &\equiv \mathbf{Y} f\ 2\,4 \\
&\equiv ((\lambda h.(\lambda x.h(x\,x))\ (\lambda x.h(x\,x)))f)\ 2\,4 \\
&\to_\beta ((\lambda x.f(x\,x))\ (\lambda x.f(x\,x)))\ 2\,4 \\
&\equiv (k\,k)\ 2\,4, \ \ \text{where}\ k \equiv \lambda x.f(x\,x) \\
&\to_\beta (f(k\,k))\ 2\,4 \\
&\equiv ((\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 \\
&\to_\beta (\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 \\
&\to_\beta^* \text{if}\ (=2\,4)\ 2\ (\text{if}\ (>2\,4)\ ((k\,k)\ (-\,2\,4)\ 4)\ ((k\,k)\ (-\,4\,2)\ 2)) \\
&\equiv (\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)) \\
&\to_\beta^* (=2\,4)\ 2\ (\text{if}\ (>2\,4)\ ((k\,k)\ (-\,2\,4)\ 4)\ ((k\,k)\ (-\,4\,2)\ 2)) \\
&\to_\delta F\ 2\ (\text{if}\ (>2\,4)\ ((k\,k)\ (-\,2\,4)\ 4)\ ((k\,k)\ (-\,4\,2)\ 2)) \\
&\equiv (\lambda x.\lambda y.y)\ 2\ (\text{if}\ (>2\,4)\ ((k\,k)\ (-\,2\,4)\ 4)\ ((k\,k)\ (-\,4\,2)\ 2)) \\
&\to_\beta^* \text{if}\ (>2\,4)\ ((k\,k)\ (-\,2\,4)\ 4)\ ((k\,k)\ (-\,4\,2)\ 2) \\
&\to \ldots \\
&\to (k\,k)\ (-\,4\,2)\ 2 \\
&\equiv ((\lambda x.f(x\,x))k)\ (-\,4\,2)\ 2 \\
&\to_\beta (f(k\,k))\ (-\,4\,2)\ 2 \\
&\equiv ((\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 \\
&\to_\beta (\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 \\
&\to_\beta^* \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 (\lambda c.\lambda t.\lambda e.c\ t\ e) \\
&\ (=(-\,4\,2)\ 2)\ (-\,4\,2)\ (\text{if}\ (>(-\,4\,2)\ 2)\ ((k\,k)\ (-\ (-\,4\,2)\ 2)\ 2)\ ((k\,k)\ (-\,2\ (-\,4\,2))\ (-\,4\,2))) \\
&\to_\beta^* (=(-\,4\,2)\ 2)\ (-\,4\,2)\ (\text{if}\ (>(-\,4\,2)\ 2)\ ((k\,k)\ (-\ (-\,4\,2)\ 2)\ 2)\ ((k\,k)\ (-\,2\ (-\,4\,2))\ (-\,4\,2))) \\
&\to_\delta (=2\,2)\ (-\,4\,2)\ (\text{if}\ (>(-\,4\,2)\ 2)\ ((k\,k)\ (-\ (-\,4\,2)\ 2)\ 2)\ ((k\,k)\ (-\,2\ (-\,4\,2))\ (-\,4\,2))) \\
&\to_\delta T\ (-\,4\,2)\ (\text{if}\ (>(-\,4\,2)\ 2)\ ((k\,k)\ (-\ (-\,4\,2)\ 2)\ 2)\ ((k\,k)\ (-\,2\ (-\,4\,2))\ (-\,4\,2))) \\
&\equiv (\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))) \\
&\to_\beta^* (-\,4\,2) \\
&\to_\delta 2
\end{aligned}$$

Figure 10.4 **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.

$$\begin{aligned}
\text{cdr}(\text{cons}\ A\ B) &\equiv (\lambda l.l\ \text{select\_second})\ (\text{cons}\ A\ B)\\
&\rightarrow_\beta (\text{cons}\ A\ B)\ \text{select\_second}\\
&\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ \text{select\_second}\\
&\rightarrow_\beta^* (\lambda x.x\ A\ B)\ \text{select\_second}\\
&\rightarrow_\beta \text{select\_second}\ A\ B\\
&\equiv (\lambda x.\lambda y.y)\ A\ B\\
&\rightarrow_\beta^* B
\end{aligned}$$

$$\begin{aligned}
\text{null? nil} &\equiv (\lambda l.l\ (\lambda x.\lambda y.\text{select\_second}))\ \text{nil}\\
&\rightarrow_\beta \text{nil}\ (\lambda x.\lambda y.\text{select\_second})\\
&\equiv (\lambda x.\text{select\_first})\ (\lambda x.\lambda y.\text{select\_second})\\
&\rightarrow_\beta \text{select\_first}\\
&\equiv T
\end{aligned}$$

$$\begin{aligned}
\text{null? }(\text{cons}\ A\ B) &\equiv (\lambda l.l\ (\lambda x.\lambda y.\text{select\_second}))\ (\text{cons}\ A\ B)\\
&\rightarrow_\beta (\text{cons}\ A\ B)\ (\lambda x.\lambda y.\text{select\_second})\\
&\equiv ((\lambda a.\lambda d.\lambda x.x\ a\ d)\ A\ B)\ (\lambda x.\lambda y.\text{select\_second})\\
&\rightarrow_\beta^* (\lambda x.x\ A\ B)\ (\lambda x.\lambda y.\text{select\_second})\\
&\rightarrow_\beta (\lambda x.\lambda y.\text{select\_second})\ A\ B\\
&\rightarrow_\beta^* \text{select\_second}\\
&\equiv F
\end{aligned}$$

■

EXAMPLE 10.65
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:

$$\text{compose} \equiv \lambda f.\lambda g.\lambda x.f\ (g\ x)$$

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 10.66
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:

$$\text{paired\_compose} \equiv \lambda p.\lambda x.(\text{car}\ p)\ ((\text{cdr}\ p)\ x)$$

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 10.5:

$$\text{curry} \equiv \lambda f.\lambda a.\lambda b.f\,(\text{cons}\ a\ b)$$

■

✓ **CHECK YOUR UNDERSTANDING**

22. What is the difference between *partial* and *total* functions? Why is the difference important?
23. What is meant by the *function space* $A \rightarrow B$?
24. Define *beta reduction*, *alpha conversion*, *eta reduction*, and *delta reduction*.
25. How does beta reduction in lambda calculus differ from lazy evaluation of arguments in a nonstrict programming language like Haskell?
26. Explain how lambda expressions can be used to represent Boolean values and control flow.
27. What is *beta abstraction*?
28. What is the **Y** combinator? What useful property does it possess?
29. Explain how lambda expressions can be used to represent structured values such as lists.
30. State the *Church-Rosser theorem*.

10 Functional Languages

10.9 Exercises

10.18 In Figure ◎10.4 we evaluated our expression in normal order. Did we really have any choice? What would happen if we tried to use applicative order?

10.19 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.

10.20 Given the definition of structures (lists) on page ◎244, what happens if we apply car or cdr to nil? How might you introduce the notion of "type error" into lambda calculus?

10.21 Let

$$\text{zero} \equiv \lambda x.x$$
$$\text{succ} \equiv \lambda n.(\lambda s.(s\ \text{select\_second})\ n)$$

where select_second $\equiv \lambda x.\lambda y.y$. Now let

$$\text{one} \equiv \text{succ zero}$$
$$\text{two} \equiv \text{succ one}$$

Show that

$$\text{one select\_second} = \text{zero}$$
$$\text{two select\_second select\_second} = \text{zero}$$

In general, show that

$$\text{succ}^n\ \text{zero select\_second}^n = \text{zero}$$

Use this result to define a predecessor function pred. You may ignore the issue of the predecessor of zero.

Note that our definitions of T and F allow us to check whether a number is equal to zero:

$$\text{iszero} \equiv \lambda n.(n\ \text{select\_first})$$

Using succ, pred, iszero, and if, show how to define plus and times recursively. These definitions could of course be made nonrecursive by means of beta abstraction and **Y**.

10 Functional Languages

10.10 Explorations

10.28 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].

10.29 Learn more about *fixed points*. We mentioned these when presenting the **Y** combinator in Section ©10.6.2. They also arise in the denotational definition of loop constructs, in metacircular interpreters (Example 10.20), and in the *data flow analysis* used by optimizing compilers (Section ©16.4.2). What do these subjects have in common? Are there important differences as well?

10.30 Explore the connection between lexical scoping in Scheme 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 does not? Is there any analogy in lambda calculus to the dynamic scoping of early dialects of Lisp?

11 Logic Languages

11.3 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 ($\land$), or ($\lor$), 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 11.39
Propositions

$$\forall C[\mathsf{rainy}(C) \land \mathsf{cold}(C) \rightarrow \mathsf{snowy}(C)]$$

(For all cities C, if C is rainy and C is cold, then C is snowy.)

$$\forall A, \forall B[(\exists C[\mathsf{takes}(A, C) \land \mathsf{takes}(B, C)]) \rightarrow \mathsf{classmates}(A, B)]$$

(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.)

$$\forall N[(N > 2) \rightarrow \neg(\exists A, \exists B, \exists C[A^N + B^N = C^N])]$$

(This is Fermat's last theorem.) ■

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 this book.

EXAMPLE 11.40
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,

$$\begin{aligned}(P_1 \rightarrow P_2) &\equiv (\neg P_1 \vee P_2)\\ (\neg\exists X[P(X)]) &\equiv (\forall X[\neg P(X)])\\ \neg(P_1 \wedge P_2) &\equiv (\neg P_1 \vee \neg P_2)\end{aligned}$$

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 below. ■

11.3.1 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 11.41
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 11.3.3. 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:

$$\mathsf{student}(A) \vee (\neg\mathsf{dorm\_resident}(A) \wedge (\neg\mathsf{takes}(A, B) \vee \neg\mathsf{class}(B)))$$

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:

$$(\mathsf{student}(A) \vee \neg\mathsf{dorm\_resident}(A)) \wedge (\mathsf{student}(A) \vee \neg\mathsf{takes}(A, B) \vee \neg\mathsf{class}(B))$$

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 11.42
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:

$$\begin{aligned} & (\mathsf{student}(A) \leftarrow \neg(\neg\mathsf{dorm\_resident}(A))) \\ & \qquad \wedge (\mathsf{student}(A) \leftarrow \neg(\neg\mathsf{takes}(A, B) \vee \neg\mathsf{class}(B))) \\ \equiv\ & (\mathsf{student}(A) \leftarrow \mathsf{dorm\_resident}(A)) \\ & \qquad \wedge (\mathsf{student}(A) \leftarrow (\mathsf{takes}(A, B) \wedge \mathsf{class}(B))) \end{aligned}$$

These are Horn clauses. The translation to Prolog is trivial:

```
student(A) :- dorm_resident(A).
student(A) :- takes(A, B), class(B).
```
■

11.3.2 Limitations

We claimed at the beginning of Section 11.1 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 $\leftarrow$ 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 11.43
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

$$\mathsf{animal}(X) \vee \mathsf{plant}(X) \vee \neg\mathsf{living}(X)$$

or equivalently

$$\text{animal}(X) \vee \text{plant}(X) \leftarrow \text{living}(X)$$

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 11.44
Empty left-hand side

As an example of an empty head, consider Fermat's last theorem (Example 11.39). 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:

$$\neg\text{big}(N) \vee \neg\text{works}(A, B, C, N)$$

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 11.45
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. ■

11.3.3 Skolemization

EXAMPLE 11.46
Skolem constants

In Example ©11.41 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`. ■

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:

EXAMPLE 11.47
Skolem functions

$$\forall X[\neg\mathsf{dorm\_resident}(X) \lor \exists A[\mathsf{campus\_address\_of}(X, A)]]$$

(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 11.48
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?

11 Logic Languages

11.6 Exercises

11.18 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).
```

11.19 Consider the following statement in predicate calculus:

$$\text{empty\_class}(C) \leftarrow \neg\exists X[\text{takes}(X, C)]$$

(a) Translate this statement to clausal form.

(b) Can you translate the statement into Prolog? Does it make a difference whether you're allowed to use \+?

(c) How about the following:

$$\text{takes\_everything}(X) \leftarrow \forall C[\text{takes}(X, C)]$$

Can this be expressed in Prolog?

11.20 Consider the seemingly contradictory statement

$$\neg\text{foo(X)} \rightarrow \text{foo(X)}$$

Convert this statement to clausal form, and then translate into Prolog. Explain what will happen if you ask

```
?- foo(bar).
```

Now consider the straightforward translation, without the intermediate conversion to clausal form:

```
foo(X) :- \+(foo(X)).
```

Now explain what will happen if you ask

```
?- foo(bar).
```

11 Logic Languages

11.7 Explorations

11.26 In Section ©11.3.1 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?

11.27 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?

11.28 *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?

11.29 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?

12 Concurrency

12.5 Message Passing

While shared-memory concurrent programming is common on small-scale multiprocessors, most concurrent programming on large multicomputers and networks is currently based on messages. In Sections ©12.5.1 through ©12.5.3 we consider three principal issues in message-based computing: naming, sending, and receiving. In Section ©12.5.4 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, Occam, and SR programming languages, the Java network library, and the MPI library package.

12.5.1 Naming Communication Partners

EXAMPLE 12.50
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 ©12.20. ■

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 12.51
`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 12.2.3, an Ada task resembles a module; its entries resemble subroutine headers nested directly inside the task.

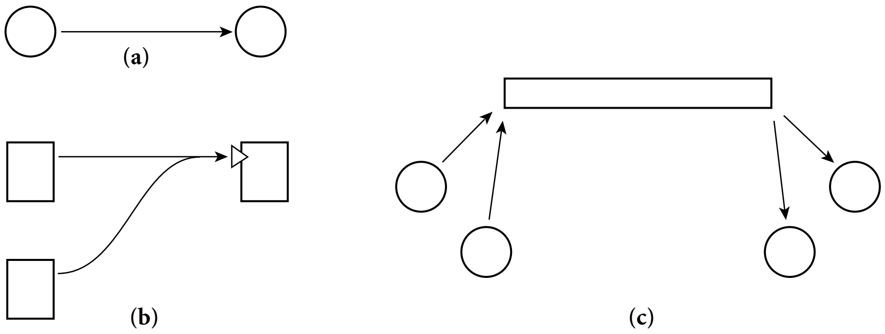


Figure 12.20 **Three common schemes to name communication partners.** In (a), processes name each other explicitly. In (b), senders name an *input port* of a receiver. The port may be called an *entry* or an *operation*. The receiver is typically a module with one or more threads inside. In (c), senders and receivers both name an independent *channel* abstraction, which may be called a *connection* or a *mailbox*.

A task receives a message that has been sent to one of its entries by executing an `accept` statement (to be discussed in Section 12.5.3). Every `entry` belongs to exactly one task; all messages sent to the same `entry` must be received by that one task. ■

EXAMPLE 12.52 Channels in Occam

The third naming option—addressing messages to channels—appears in Occam. (Though based on CSP, Occam differs from it in several concrete ways, including the use of channels.) Channel declarations are supported with the built-in `CHAN` and `CALL` types:

```
CHAN OF BYTE stream :
CALL lookup(RESULT [36]BYTE name, VAL INT ssn) :
```

These declarations specify a one-directional channel named `stream` that carries messages of type `BYTE` and a two-directional channel named `lookup` that carries requests containing an integer named `ssn` and replies containing a 36-byte string named `name`. `CALL` channels are syntactic sugar for a pair of `CHAN` channels, one in each direction. To send a message on a `CHAN` channel, an Occam thread uses a special "exclamation point" operator:

```
stream ! 'x'
```

To send a message (and receive a reply) on a `CALL` channel, a thread uses syntax that resembles a subroutine call:

```
lookup(name, 123456789)
```
■

We noted in our coverage of parallel loops (page 591) that language rules in Occam prohibit concurrent threads from making conflicting accesses to the same variable. For channels, the basic rule is that exactly one thread may send to a channel, and exactly one may receive from it. (For `CALL` channels, exactly one thread may send requests, and exactly one may accept them and send replies.) These rules are relaxed in Occam 3 to permit `SHARED` channels, which provide a mutual exclusion mechanism. Only one thread may accept requests over a `SHARED CALL` channel, but multiple threads may send them. In a similar vein, multiple threads may `CLAIM` a set of `CHAN` channels for exclusive use in a critical section, but only one thread may `GRANT` those channels; it serves as the other party for every message sent or received.

In SR and the Internet libraries of Java we see combinations of our naming options. An SR program executes on a collection of one or more *virtual machines*,[1] each of which has a separate address space, and may be implemented on a separate node of a network. Within a virtual machine, messages are sent to (and received from) a channel-like abstraction called an `op`. Unlike an Occam channel, an SR `op` has no restrictions on the number or identity of sending and receiving threads: any thread that can see an `op` under the usual lexical scoping rules can send to it or receive from it. A `receive` operation must name its `op` explicitly; a `send` operation may do so also, or it may use a *capability* variable. A capability in SR is like a pointer to an `op`, except that pointers work only within a given virtual machine, while capabilities work across the boundaries between them. Aside from start-up parameters and possibly I/O, capabilities provide the *only* means of communicating among separate virtual machines. At the outer-most level, then, an SR program can be seen as having a port-like naming scheme: messages are sent (via capabilities) to `op`s of virtual machines, within which they may potentially be received by any local thread.

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 ◎12.20b): each message is sent to a specific *Internet address* and *port number*.[2] The TCP protocol also uses port-based naming, but only for the purpose of establishing *connections* (Figure ◎12.20c), which it then

1 These are unrelated to the notion of virtual machine discussed in Section 15.1.

2 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, as of 2008 most addresses are still 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.

uses for all subsequent communication. Connections deliver messages reliably and in order.

EXAMPLE 12.53
Datagram messages in Java

To send or receive UDP messages, a Java thread must create a *datagram socket*:

```
DatagramSocket my_socket = new DatagramSocket(port_id);
```

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 my_msg = new DatagramPacket(buf, len, addr, port);
...  // initialize message
my_socket.send(my_msg);
```

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:

```
my_socket.receive(my_msg);
...  // parse content of my_msg
```

■

EXAMPLE 12.54
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 my_server_socket = new ServerSocket(port_id);
Socket client_connection = my_server_socket.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 server_connection = new Socket(host_name, port_id);
```

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 ©7.9.3):

```
BufferedReader in = new BufferedReader(
    new InputStreamReader(client_connection.getInputStream()));
PrintStream out =
    new PrintStream(client_connection.getOutputStream());
// This is in the server; the client would make streams out
// of server_connection.
...
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 (though as we saw in SR, if a receiver is a complex entity like a virtual machine, it may have many threads inside). 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 different orders.[3] When naming channels, a path links all the senders that can 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. An SR `op` can be used by any thread to which it is visible. In both cases, the channel functions as a queue: send (enqueue) and receive (dequeue) operations are ordered, so that everything is received in the order it was sent.

12.5.2 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.

3 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 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 cables.

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 12.55
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.

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 ©12.21. ■

No-wait `send` appears in Erlang, SR, and the Java Internet library. Synchronization `send` appears in Occam. Remote-invocation `send` appears in Ada, in Occam, and in SR. 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, unfortunately, 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 changing parameters of the underlying message delivery mechanism.

EXAMPLE 12.56
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

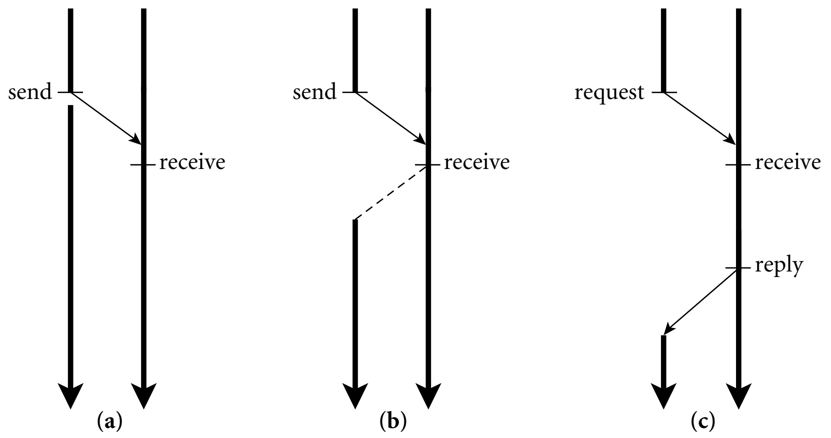


Figure 12.21 **Synchronization semantics for the `send` operation: no-wait `send` (a), synchronization `send` (b), and remote-invocation `send` (c).** In each diagram we have assumed that the original message arrives before the receiver executes its `receive` operation; this need not in general be the case.

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

DESIGN & IMPLEMENTATION

The semantic impact of implementation issues

The inability to buffer unlimited amounts of data, or to report errors synchronously to a sender that has continued execution, are only the most recent of the 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`, `downto`, and constant step sizes for `for` loops (Section 6.5.1); limits on set universe size (to accommodate bit vectors—Section 7.6); limited procedure nesting (to accommodate displays—Section 8.1); the fixed size requirement for opaque exports in Modula-2 (Section 9.2.1); and the lack of nested threads or of unrestricted arms on a `cobegin` statement (to avoid the need for cactus stacks—Section 8.6.1). 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.

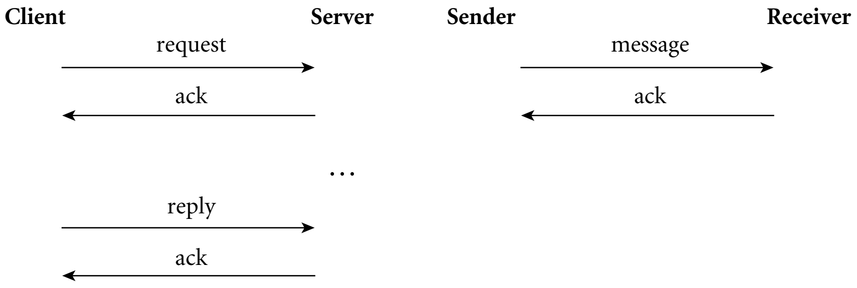


Figure 12.22 **Acknowledgment messages for error detection.** In the absence of piggy-backing, remote-invocation `send` (left) may require four underlying messages; synchronization `send` (right) may require two.

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 12.57
Acknowledgments

If the underlying message-passing system is unreliable, a language or library will typically employ *acknowledgment* messages to verify successful transmission (Figure 12.22). 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. ■

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. MPI-2 provides 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. 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 take a fixed, static number of parameters, two of which typically specify the location and size of

DESIGN & IMPLEMENTATION

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 ◎12.35 and Exploration ◎12.49.

the message to be sent. To send a message containing values held in more than one program variable, the programmer must explicitly *gather*, or *marshal*, those values into the fields of a record. On the receiving end, the programmer must *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 14.6.2). Finally, as we shall see in Section ◎12.5.3, 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.

12.5.3 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 12.2.3 (page 597). Among the languages and systems we have been using as examples, only SR provides implicit receipt (some RPC systems also provide it, as we shall see in Section ◎12.5.4).

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 must 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, Occam, and SR 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 ◎6.7.

```
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 12.23 Bounded buffer in Ada, with an explicit manager task.

EXAMPLE 12.58
Bounded buffer in Ada 83

Figure ©12.23 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 12.3.4.) The Ada `accept` statement receives the `in` and `in out` parameters (Section 8.3.1) 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 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 the sidebar on page Ⓒ160.) ■

EXAMPLE 12.59
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 one 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 12.60
Bounded buffer in Occam

Occam's equivalent of select is known as ALT. As in Ada, the choice among arms can be based both on Boolean conditions and on the availability of messages. (One minor difference: Occam semantics specify a one-step evaluation process; message availability is considered part of the guard.) The body of our bounded buffer example is shown in Figure Ⓒ12.24. Recall that Occam uses indentation to delimit control-flow constructs. Also note that Occam has no mod operator.

The question-mark operator (?) is Occam's receive; the exclamation-mark operator (!) is its send. As in Ada, an active manager thread must embed the ALT statement in a loop. As written here, the ALT statement has two guards. The first guard is true when full_slots < SIZE and a message is available on the channel named producer; the second guard is true when full_slots > 0 and a message is available on the channel named request. ■

EXAMPLE 12.61
Asymmetry of synchronization send

Because we are using synchronization send in this example, there is an asymmetry between the treatment of producers and consumers: the former need only send the manager data; the latter must send it a dummy argument and then wait for the manager to send the data back:

```
-- channel declarations:
CHAN OF BDATA producer, consumer :
CHAN OF BOOL request :

-- buffer manager:
...      -- (data declarations omitted)
WHILE TRUE
  ALT
    full_slots < SIZE & producer ? d
      SEQ
        buf[next_empty] := d
        IF
          next_empty = SIZE
            next_empty := 1
          next_empty < SIZE
            next_empty := next_empty + 1
        full_slots := full_slots + 1
    full_slots > 0 & request ? t
      SEQ
        consumer ! buf[next_full]
        IF
          next_full = SIZE
            next_full := 1
          next_full < SIZE
            next_full := next_full + 1
        full_slots := full_slots - 1
```

Figure 12.24 Bounded buffer in Occam.

```
-- producer:                     -- consumer:
producer ! x                     request ! TRUE
                                 consumer ? x
```

The asymmetry could be removed by using remote invocation on CALL channels:

```
-- channel declarations:
CALL insert(VAL BDATA d) :
CALL remove(RESULT BDATA d) :

-- buffer manager:
WHILE TRUE
  ALT
    full_slots < SIZE & ACCEPT insert(VAL BDATA d)
      buf[next_empty] := d
      IF  -- increment next_empty, etc.
      ...
    full_slots > 0 & ACCEPT remove(RESULT BDATA d)
      d := buf[next_full]
      IF  -- increment next_full, etc.
      ...
```

Client code now looks like this:

```
-- producer:                        -- consumer:
insert(x)                           remove(x)
```

In the code of the buffer manager, the body of the ACCEPT is the single subsequent statement (the one that accesses buf). Updates to next_empty, next_full, and full_slots occur after replying to the client. ■

EXAMPLE 12.62
Timeout in Occam receipt

The effect of an Ada delay can be achieved in Occam by an ALT arm that "receives" from a *timer* pseudoprocess:

```
clock ? AFTER quit_time
```

An arm can also be selected on the basis of a Boolean condition alone, without attempting to receive:

```
a > b & SKIP        -- do nothing
```

Occam's ALT has no equivalent of the Ada terminate, nor is there an else (a similar effect can be achieved with a very short delay). ■

EXAMPLE 12.63
Bounded buffer in Erlang

In Erlang, which uses no-wait send, one might at first expect asymmetry similar to that of Occam: 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 12.25. 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.
```

■

EXAMPLE 12.64
Bounded buffer in SR

In SR, selective receipt is again based on guarded commands; code appears in Figure 12.26. The st stands for "such that"; it introduces the Boolean half of a guard. Client code looks like this:

```
# producer:                         # consumer:
call insert(x)                      x := remove()
```

If desired, an explicit reply to the client could be inserted between the access to buf and the updates of next_empty, next_full, and full_slots in each arm of the in. ■

EXAMPLE 12.65
Peeking at messages in SR and Erlang

In a significant departure from Ada and Occam, both SR and Erlang place the parameters of a potential message within the scope of the guard condition, allowing a receiver to "peek inside" a message before deciding whether to receive it. In SR, we can say

```
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 12.25 **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 Occam. Each clause of the `receive` ends with a tail recursive call.

```
in insert(d) st d % 2 = 1 ->            # only accept odd numbers
```

In Erlang,

```
receive
    {insert, D} when D rem 2 == 1 ->    % likewise
```

In SR, a receiver can also accept messages on a given port (i.e., of a given op) out of order, by specifying a *scheduling expression*:

```
in insert(d) st d % 2 = 1 by -d ->
    # only accept odd numbers, and pick the largest one first
```

■

Like an Ada `select`, an SR `in` statement can end with an `else` guard; this guard will be selected if no message is immediately available. There is no equivalent of `delay` or `terminate`.

DESIGN & IMPLEMENTATION

Peeking inside messages

The ability of guards and scheduling expressions to "peek inside" a message in SR and Erlang requires that all pending messages be visible to the language run-time system. An SR 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.

```
resource buffer
  op insert(d : bdata)
  op remove() returns d : bdata
body buffer()
  const SIZE := 10;
  var buf[0:SIZE-1] : bdata
  var full_slots := 0, next_empty := 0, next_full := 0
  process manager
    do true ->
      in insert(d) st full_slots < SIZE ->
          buf[next_empty] := d
          next_empty := (next_empty + 1) % SIZE
          full_slots++
      [] remove() returns d st full_slots > 0 ->
          d := buf[next_full]
          next_full := (next_full + 1) % SIZE
          full_slots--
      ni
    od
  end  # manager
end  # buffer
```

Figure 12.26 Bounded buffer as an active SR process.

12.5.4 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 8.3.1, 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 ◎8.3.2), but only at very high cost. As noted in Section 7.10, a few languages and systems perform deep copies of linked data structures passed to remote routines.

Performance differences between local and remote calls can only be hidden 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.

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 a language like SR) must bridge this explicit-to-implicit gap. The typical implementation resembles the thread-based event handling of Section 8.7.2. 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 12.66
An RPC server system

Figure Ⓒ12.27 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 generator. 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 threads are implemented on top of one OS process or several.

In the simplest case—a single-threaded server on a single OS process—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. ■

DESIGN & IMPLEMENTATION

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 processors of a multicomputer or 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*: arguments become uninitialized from the caller's point of view, and can therefore migrate to a remote callee without danger of inducing remote references.

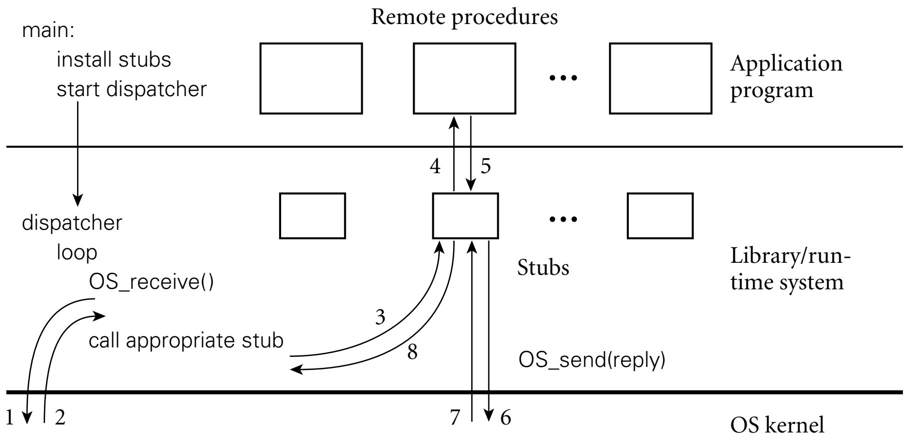


Figure 12.27 **Implementation of a remote procedure call server.** Application code initializes the RPC system by installing stubs generated by the stub compiler (not shown). It then calls into the run-time system to enable incoming calls. Depending on details of the particular system in use, the dispatcher may use the main program's single process (in which case the call to start the dispatcher never returns), or it may create a pool of processes that handle incoming requests.

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 12.2.4, 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 multiprocess server, the call to start the dispatcher will generally ask the kernel to fork a "pool" of processes to service remote requests. Each of these processes will then perform the operations described in the previous paragraphs. In a language or library with a one–one correspondence between threads and processes, each process 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 process will run threads from the ready list until the list is empty, at which point it (the process) will call into the kernel for another message. So long as the number of runnable threads is greater than or equal to the number of processes, no new messages will be received. When the number of runnable threads drops below the number of processes, then the extra processes will call into the kernel, where they will block until requests arrive.

✓ CHECK YOUR UNDERSTANDING

50. Describe three ways in which processes 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, equivalently, of `ALT` in Occam).
60. What semantic and pragmatic challenges are introduced by the ability to "peek" inside messages before they are received?

12 Concurrency

12.7 Exercises

12.33 In Section 12.4.1 we cast monitors as a mechanism for synchronizing access to shared memory, and we described their implementation in terms of semaphores. It is also possible to think of a monitor as a module inhabited by a single process, which accepts request messages from other processes, 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 12.2.3, page 597.)

12.34 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.

12.35 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 [Tan02, PD07].)

12.36 An arm of an Occam `ALT` statement may include an *input guard*—a receive (`?`) operation—in which case the arm can be chosen only if a potential partner is trying to send a matching message. One could imagine allowing *output guards* as well: send (`!`) operations that would allow their arm to be chosen only if a potential partner were trying to receive a matching message. Neither Occam nor CSP (as originally defined) permits output guards. Can you guess why? Suppose you wished to provide them. How would the implementation work? (Hint: For ideas, see the articles of

Bernstein [Ber80], Buckley and Silbershatz [BS83b], Bagrodia [Bag86], or Ramesh [Ram87].)

12.37 In Section ◎12.5.3 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, Occam, and SR 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, Occam, and SR? (Hint: For ideas, see the work of Apt and Francez [Fra80, AF84].)

12 Concurrency

12.8 Explorations

12.49 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`?

12.50 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?

12.51 Language designers and concurrency experts have argued for more than 30 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?

13 Scripting Languages

13.3.5 XSLT

HTML was inspired by an older standard known as SGML (standard generalized markup language), widely used in the business world to represent structured data. Because it evolved in such an ad hoc way, HTML has been very difficult to standardize. Incompatibilities among browsers continue to frustrate web designers, and several features of the language that have been deprecated 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 13.83
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 frequently 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 more and more vendors work to bring web content to cell phones, televisions, handheld computers, and audio-only devices, the need to distinguish between content and presentation is becoming increasingly critical. SGML has always made this distinction, but it is widely seen as overkill: far too complex for use on the web. ■

This is where XML steps in. XML (extensible markup language) is a deliberately streamlined descendant of SGML with at least three important advantages over HTML: (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). As noted in the main text, 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. It helps to remember the three families of markup languages—SGML, HTML, and XML—and to know that each 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 on stylesheets, but web sites increasingly use CSS to create a uniform "look and feel" across a collection of pages without embedding redundant information in every page.

SGML and DSSSL remain important in the business world, but are little used on the web. HTML is likely to persist for a very long time, but its lack of extensibility and its mix of content and presentation are increasingly perceived as fundamental limitations. XML is widely viewed as the notation of the future. Even for documents that remain in HTML, designers are likely to migrate toward XHTML (extensible hypertext markup language), an almost (but not quite) backward compatible variant of HTML that conforms to the XML standard.

XML and XHTML

EXAMPLE 13.84
Well-formed XHTML

An XML document must be *well formed*: tags must either constitute properly nested, matched pairs, or be explicit singletons, which end with a "`/>`" delimiter. The following fragment, for example, is well-formed (though incomplete) XHTML:

```
<em><q><a id="favorite-quote" />I defy the tyranny of precedent</q>
(Clara Barton).</em>
```

Here the quotation element (`<q>...</q>`) is nested inside the emphasis element (`<em>...</em>`). Moreover the anchor element (`<a.../>`), which can serve as the target of a link, 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, or if the opening `<em><q>` tags were reversed (`<q><em>`). ■

Well-formedness is a simple syntactic rule, like the requirement that parentheses be balanced in Lisp. It makes XML (and thus XHTML) much easier than plain HTML to parse and to process automatically. The careful reader may also have noticed that we used lower-case letters for tags in XHTML, where previous HTML examples were all in upper case. HTML is case-insensitive; either style is accepted, though upper case has been the convention in standards documents. XML is case-sensitive, so `<em>` and `<EM>` are different. The XHTML designers had to pick one. Going against the existing convention (but not the existing rules) preserves backward compatibility while helping the reader identify documents that are likely to conform to the newer standard.

The set of tags to be used in an XML document is specified by either a *document type definition* (DTD) or an *XML Schema*. DTDs are inherited from SGML. They indicate which tags are allowed, whether they are pairs or singletons, whether they permit attributes (name-value pairs like the `id="favorite-quote"` in Example ◎13.84), and whether any attributes are mandatory. The rules of the DTD take the form of XML *declarations*, which look like elements beginning with a "`<!`" delimiter. These can be included directly in the XML document. More often they are kept in an external document with its own URI, and the XML document begins with a `<!DOCTYPE ... >` declaration that specifies that URI. (Comments also look like declarations: `<!-- ignored -->`.) If an XML document has no explicit DTD (neither in-line nor external), it is said to define a DTD *implicitly* by virtue of which tags are actually used.

XML Schemas are a newer mechanism, meant to replace DTDs. They are written in XSD, the XML Schema Definition language, which is itself an example of well-formed XML, defined by a DTD. Because they are written in XSD, XML Schemas can be created using XML-aware editors, parsed with XML parsers, and transformed with XSLT. In comparison to DTDs, XSD provides a significantly richer vocabulary for specifying syntactic rules. Among other things, it allows the designer to specify the data types of elements and attributes in considerable detail, providing a level of automatic checking not possible with DTDs. XSD also supports inheritance, so one XML Schema can be defined as an extension of another. As of this writing, DTDs remain more common than XML Schemas. In particular, the XML Schema for XHTML did not became official until 2008. We will rely on DTDs in the remainder of this section.

EXAMPLE 13.85

XHTML to display a favorite quote

Because tags must nest in XML, a document has a natural tree-based structure. Figure ◎13.24 shows the source for a small but complete XHTML 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 (non-leaf) 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.

Our document begins with an `<?xml ... ?>` declaration, which indicates the version of XML and the character encoding used in the rest of the document. The declaration is included for the benefit of tools that process the document; it isn't part of the XML source itself. (Note the syntactic resemblance to the *processing instructions* used in Section 13.3.2 to provide input to the PHP interpreter.)

The second line of our document is a `<!DOCTYPE ... >` declaration that names an XHTML DTD at the World Wide Web Consortium. The remainder of the document is data. The root, named "`/`", has one child: the `html` element. This in turn has two children: the `head` and the `body`. The `head` has a `title` child and an `xmlns` attribute. The latter declares `xhtml` to be the default *namespace* for the document. Namespaces in XML are similar to the namespaces of C++ or the packages of Java (Section 3.8); they allow us to give tag names a disambiguating 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. ■

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>Favorite Quote</title>
</head>
<body>
<p>
<em><q><a id="favorite-quote" />
I defy the tyranny of precedent</q>
(Clara Barton).</em>
</p>
</body>
</html>
```

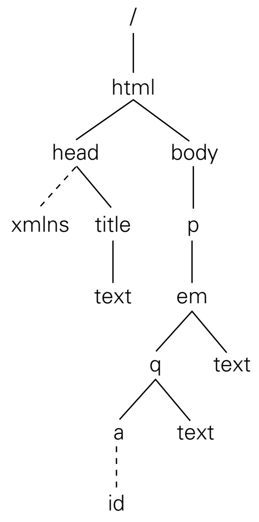


Figure 13.24 A complete XHTML document and its corresponding tree. Child relationships are shown with solid lines, attributes with dashed lines.

XSLT, XPath, and XSL-FO

XSL (extensible stylesheet language) can be thought of as a language for specifying what to *do* with an XML document. It has three sublanguages, called XSLT, XPath, and XSL-FO. 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 13.86
XPath names for XHTML elements

XPath is a language used to name things in XML files. XPath names frequently appear in the attributes of XSLT elements. Returning to Figure 13.24, the quotation element of our document could be named in XPath as `/html/body/p/em/q`. The quotation element and its text-node sibling, together, could be be named as `/html/body/p/em/*`. 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 document intended for the web into a multipage document intended for printing on paper. For the sake of simplicity, we will not use XSL-FO in any of our examples. Rather we will format XML documents by using XSLT to turn them into HTML.

```
<?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</title>
    <publisher>Network Theory, Ltd.</publisher>
    <address>Bristol, UK</address>
    <year>2003</year>
    <note>Available at <uri>http://www.network-theory.co.uk/docs/pylang/</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>http://www.mulberrytech.com/Extreme/Proceedings/html
/2004/Kepser01/EML2004Kepser01.html</uri></note>
  </inproceedings>
```

Figure 13.25 **A bibliography in XML.** References (two books, a journal article, and three conference papers) appear in arbitrary order. The Kepser URI has 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>Larry Wall</author>
    <author>Tom Christiansen</author>
    <author>Jon Orwant</author>
    <title>Programming Perl</title>
    <edition>third</edition>
    <publisher>O&#8217;Reilly and Associates</publisher>
    <address>Cambridge, MA</address>
    <year>2000</year>
  </book>
</bibliography>
```

Figure 13.25 *(continued)*

An XML document can explicitly specify an XSLT script that should be used to transform or format it. This is a standard 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. Unfortunately, as of this writing the details vary across both server and client platforms.

Extended Example: Bibliographic Formatting

EXAMPLE 13.87
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 13.25 contains XML source for 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.

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 the sidebar on page 295.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://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 13.26 Bibliography stylesheet in XSL. This script will generate HTML when applied to a bibliography like that of Figure 13.25. *(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(., 'http://')"/></a>
</xsl:template>

<xsl:template match="@*|node()">        <!-- default: copy content -->
  <xsl:copy><xsl:apply-templates select="@*|node()"/></xsl:copy>
</xsl:template>

</xsl:stylesheet>
```

Figure 13.26 *(continued)*

Figure ◎13.26 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 ◎13.25). 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 `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` 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 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 *http://* 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 instruction 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

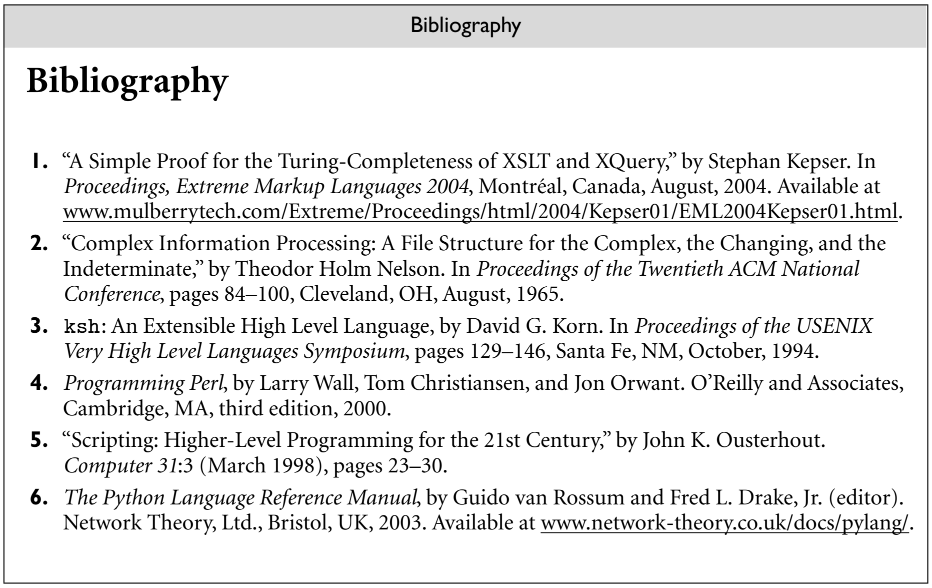
```
<html><head><title>Bibliography</title></head>
<body><h1>Bibliography</h1><ol>
<li>
  <q>A Simple Proof for the Turing-Completeness of XSLT and XQuery,</q>
  by Stephan Kepser.&nbsp; In <em>Proceedings, Extreme Markup Languages
  2004</em>, Montr&eacute;al, Canada, August, 2004.  Available at
  <a href="http://www.mulberrytech.com/Extreme/Proceedings/html/2004/Kepser01
/EML2004Kepser01.html">www.mulberrytech.com/Extreme/Proceedings/html/2004
/Kepser01/EML2004Kepser01.html</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 Larry Wall, Tom Christiansen, and Jon
  Orwant.&nbsp; O&rsquo;Reilly and Associates, Cambridge, MA, third edition,
  2000.</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,</em> by Guido van Rossum and
  Fred L. Drake, Jr. (editor).&nbsp; Network Theory, Ltd., Bristol, UK, 2003.
  Available at <a href="http://www.network-theory.co.uk/docs/pylang/">www.network-
theory.co.uk/docs/pylang/</a>.</li>
</ol>
</body></html>
```

Figure 13.27 Result of applying the stylesheet of Figure 13.26 to the bibliography of Figure 13.25.

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 13.27. The rendered web page appears in Figure 13.28.

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. ■



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 www.mulberrytech.com/Extreme/Proceedings/html/2004/Kepser01/EML2004Kepser01.html.
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 Larry Wall, Tom Christiansen, and Jon Orwant. O’Reilly and Associates, Cambridge, MA, third edition, 2000.
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*, by Guido van Rossum and Fred L. Drake, Jr. (editor). Network Theory, Ltd., Bristol, UK, 2003. Available at www.network-theory.co.uk/docs/pylang/.

Figure 13.28 Rendered version of the HTML in Figure 13.27.

CHECK YOUR UNDERSTANDING

58. Explain the relationships among SGML, HTML, and XML. What are their corresponding stylesheet languages?
59. Why does XML work so hard to distinguish between *content* and *presentation*?
60. What are the three main components of XSL? What are their respective purposes?
61. What is XHTML? How does it differ from HTML?
62. Explain the correspondence between XML documents and trees.
63. What does it mean for an XML document to be *well formed*?
64. What is a *document type definition* (DTD)? An *XML Schema*? Briefly, how do they compare?
65. Explain the distinctions (syntactic and semantic) among *elements*, *declarations*, and *processing instructions* in XML. Also explain the distinctions among *elements*, *tags*, and *attributes*.
66. Summarize the execution model of XSLT. In a nutshell, how does it work?
67. Explain the difference between *applying* templates and *calling* them in XSLT.

13 Scripting Languages

13.6 Exercises

13.19 Modify the XSLT of Figure ©13.26 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.

(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 [Uni03] or the ACM Transactions [*www.acm.org/pubs/submissions/latex_style/index.htm*]).

13.20 Suppose bibliographic entries in Figure ©13.25 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).

13.21 Write a program that will read an XHTML file and print an outline of its contents, by extracting all `<title>`, `<h1>`, `<h2>`, and `<h3>` elements, and printing them at varying levels of indentation. Write

(a) in C or Java

(b) in `sed` or `awk`

(c) in Perl, Python, Tcl, or Ruby

(d) in XSLT

Compare and contrast your solutions.

13 Scripting Languages

13.7 Explorations

13.31 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?

13.32 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.

13.33 Learn about XSL-FO. Use it to reimplement Example ©13.87. 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 Building a Runnable Program

14.2 Intermediate Forms

In this section we consider three widely used intermediate forms: Diana, GIMPLE, and RTL. Two additional examples, Java byte code and the Common Intermediate Language (CIL) can be found in Chapter 15.

Diana (Descriptive Intermediate Attributed Notation for Ada) is an Ada-specific, high-level tree-based IF developed cooperatively by researchers at the University of Karlsruhe in Germany, Carnegie Mellon University, Intermetrics, Softech, and Tartan Laboratories. It incorporates features from two earlier efforts, named TCOL and AIDA.

GIMPLE and RTL are the intermediate languages of the GNU compiler collection (`gcc`). RTL (Register Transfer Language) is the older of the two. It is a medium-level pseudo-assembly language, and was the basis of almost all language-independent code improvement in `gcc` prior to the introduction of GIMPLE in 2005. GIMPLE, like Diana, is a tree-based form, but not quite as abstract. As of `gcc` v.4, there are approximately 100 code improvement phases based on GIMPLE, and about 70 based on RTL.

14.2.1 Diana

Diana is very complex (the documentation is 200 pages long), but highly regular, and we can at least give the flavor of it here. It is formally described using a preexisting notation called IDL [SS89], which stands for Interface Description Language.[1] IDL is widely used to describe abstract data types in a machine- and implementation-independent way. Using IDL-based tools, one can automatically construct routines to translate concrete instances of an abstract data type to and

1 Unfortunately, the term “IDL” is used both for the general category of interface description languages (of which there are many) and the specific Interface Description Language used by Diana.

from a standard linear textual representation. IDL is perfectly suited for Diana. Other uses include multi-database systems, message passing across distributed networks, and compilation for heterogeneous parallel machines. In addition to providing the interface between the front end and back end of an Ada compiler, Diana frequently serves as the standard representation of fragments of Ada code in a wider program development environment.

Diana structures are defined abstractly as trees, but they are not necessarily represented that way. To guarantee portability across platforms and among the products produced by different vendors, all programs that use Diana must be able to read and write the linear textual format. Vendors are allowed (and in fact encouraged) to extend Diana by adding new attributes to the tree nodes, but a tool that produces Diana conforming to the standard must generate all the standard attributes and must never use the standard attributes for nonstandard purposes. Similarly, a tool that consumes Diana conforming to the standard may exploit information in extra attributes if it is provided, but must be capable of functioning correctly when given only the standard attributes.

Ada compilers construct and decorate the nodes of a Diana tree in separate passes. The Diana manual recommends that the construction pass be driven by an attribute grammar. This pass establishes the lexical and syntactic attributes of tree nodes. Lexical attributes include the spelling of identifier names and the location (file name, line and column number) of constructs. Syntactic attributes are the parent–child links of the tree itself.[2] Subsequent traversal(s) of the tree establish the semantic and code-based attributes of tree nodes. Code-based attributes represent low-level properties such as numeric precision that have been specified in the Ada source.

Symbol table information is represented in Diana as semantic attributes of declarations, rather than as a separate structure. If desired, an *implementation* of Diana can break this information out into a separate structure for convenience, so long as it retains the tree-based abstract interface. Occurrences of names are then linked to their declarations by "cross links" in the tree. A fully attributed Diana structure is therefore in fact a DAG, rather than a tree. The cross links are all among the semantic attributes, so the initial structure (formed of lexical and syntactic attributes) is indeed a tree.

IDL (and thus the Diana definition) employs a tree grammar notation similar to that of Section 4.6. Unlike BNF this notation defines a complete syntax tree, rather than just its fringe (i.e., the yield). To avoid the many "useless" nodes of a typical parse tree, IDL distinguishes between two kinds of symbols, which it calls *classes* and *nodes*. The nodes are the "interesting" symbols—the ones that are in the Diana tree. The classes are the "uninteresting" symbols; they exist to facilitate construction of the grammar. In effect, the distinction between classes and nodes

2 Terminology here is potentially confusing. We have been using the term "attribute" to refer to annotations appended to the nodes of a parse or syntax tree. Diana uses the term for *all* the information stored in the nodes of a syntax tree. This information includes the references to other nodes that define the structure of the tree.

```
Structure ExpressionTree Root EXP is
    -- ExpressionTree is the name of the abstract data type.
    -- EXP is the start symbol (goal) symbol of the grammar.

    Type Source_Position ;
        -- This is a private (implementation-dependent) type.

    EXP  ::= leaf | tree ;
        -- EXP is a class.  By convention, class names are written
        -- in all upper-case letters.  They are defined with "::="
        -- productions.  Their right-hand-sides must be an alternation
        -- of singletons, each of which is either a class or a node.

    tree  => as_op: OPERATOR,  as_left: EXP,  as_right: EXP ;
    tree  => lx_src: Source_Position ;
    leaf  => lx_name: String ; lx_src: Source_Position ;
        -- tree and leaf are nodes.  They are the symbols actually
        -- contained in an ExpressionTree.  Their attributes (including
        -- substructure) are defined by "=>" productions.  Multiple
        -- productions for the same node are NOT alternatives; they
        -- define additional attributes.  Thus, every tree node has four
        -- attributes: as_op, as_left, as_right, and lx_src.  Every leaf
        -- has two attributes: lx_name and lx_src.  By convention,
        -- Diana uses 'lx_' to preface lexical attributes,
        -- 'as_' to preface abstract syntax attributes,
        -- 'sm_' to preface semantic attributes, and
        -- 'cd_' to preface code attributes.

        -- In a more realistic example, leaf would have a sm_dec
        -- attribute that identified its declaration node, where
        -- additional attributes would describe its type, scope, etc.

    OPERATOR  ::=  plus | minus | times | divide ;
    plus => ;  minus => ;  times => ;  divide => ;
        -- OPERATOR is a class consisting of the standard four binary
        -- operators.  The null productions reflect the fact that an
        -- operator's name tells us all we need to know about it.
        -- We could have made the operator of a tree node a private
        -- type, eliminating the need for the null productions and empty
        -- subtree, but this would have pushed operators out of the
        -- machine-independent part of the notation, which is unacceptable.
End
```

Figure 14.11 Example of the IDL notation used to define Diana.

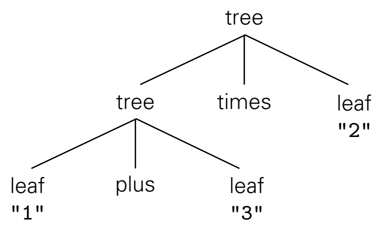


Figure 14.12 Abstract syntax tree for `(1 + 3) * 2`, using the IDL definition of Figure 14.11. Every node also has an attribute `src` of type `Source_Position`; these are not shown here.

serves the same purpose as the $A : B$ notation introduced for the left-hand sides of productions in Section 4.6 (Figure 14.6).

EXAMPLE 14.19
`ExpressionTree` abstraction in Diana

Figure 14.11 contains an IDL example adapted from the Diana manual [GWEB83, p. 26]. The `ExpressionTree` abstraction defined here is much simpler than the corresponding portion of Diana, but it serves to illustrate the IDL notation. An `ExpressionTree` for `(1 + 3) * 2` appears in Figure 14.12. Note that the classes (`EXP` and `OPERATOR`) do not appear in the tree. Only the nodes (`tree` and `leaf`) appear. ■

14.2.2 The `gcc` IFs

Many readers will be familiar with the `gcc` compilers. Distributed as open source by the Free Software Foundation, `gcc` is used very widely in academia, and increasingly in industry as well. The standard distribution includes front ends for C, C++, Objective-C and C++, Ada 95, Fortran, and Java. Front ends for additional languages, including Pascal Modula-2, PL/I, Mercury, and Cobol are separately available. The C compiler is the original, and the one most widely used (`gcc` originally stood for "GNU C compiler"). There are back ends for dozens of processor architectures, including all commercially significant options. There are also GNU implementations, not based on `gcc`, for some two dozen additional languages.

`Gcc` has three main IFs. Most of the (language-specific) front ends employ, internally, some variant of a high-level syntax tree form known as GENERIC. Early phases of machine-independent code improvement use a somewhat lower-level tree form known as GIMPLE (still a high-level IF). Later phases use a linear, medium-level IF known as RTL (register transfer language).

GIMPLE is a recent innovation. 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. As of `gcc` v.4, GENERIC is 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. The “middle end” then performs as many as 100 phases of code improvement on the GIMPLE representation, 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

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:

EXAMPLE 14.20
GCD program in GIMPLE

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

Figure 14.13 illustrates the “high GIMPLE” produced by the C front end of gcc 4.0 when given this program as input. If we compare this GIMPLE code to Figure 14.2, which loosely[3] resembles GENERIC, we see at least two significant differences. First, temporary variables have been introduced to hold the values obtained from getint (GIMPLE declines to write the result of a function call directly to an in-memory variable). Second, the while loop has been recast with explicit gotos.

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. The if statement inside the loop, for example, will see its then and else parts converted into simple gotos, which will jump to separate statements. 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*. We will study SSA in more detail in

3 Unlike the informal notation of Figure 14.2, 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.

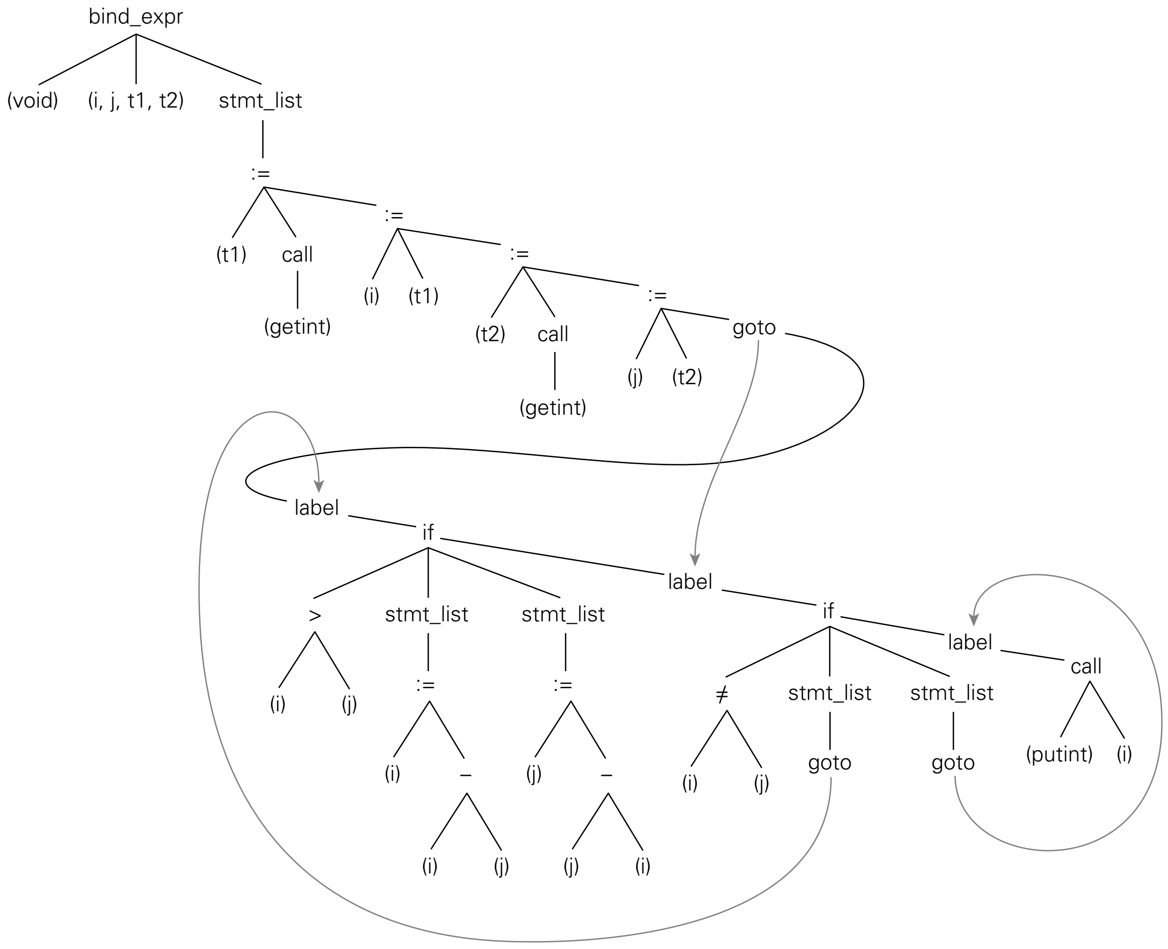


Figure 14.13 **Simplified GIMPLE for the `gcd` program.** Only structural nodes are shown: references to nodes that constitute symbol table information are indicated by parenthesized names. The node for function **`main`** would contain a pointer to the bind_expr (block) node at the root of the tree shown here.

Section Ⓒ16.4.1. Briefly, the SSA conversion introduces extra variable names into the program in such a way that nothing is ever written in more than one place. If there are 10 assignments to variable foo in the source code, there will be (at least) ten separate variables $\text{foo}_1, \ldots, \text{foo}_{10}$ in SSA. When control paths merge (e.g., after an `if... then... else`), versions of a variable arriving on different paths are combined, using a hypothetical "phi function" to create yet another version ($\text{foo}_{11} := \phi(\text{foo}_1, \text{foo}_2)$). As in functional programming languages, the single-assignment character of SSA means that expressions are *referentially transparent*—independent of evaluation order. Referential transparency significantly simplifies many forms of code improvement.

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.

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; `insns` 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 14.21
An RTL `insn` sequence

A simplified `insn` sequence for the code fragment `d := (a + b) * c` appears in Figure ◎14.14. The three leading numbers in each `insn` represent the `insn`'s unique id and those of its predecessor and successor, respectively. The `:SI` *mode specifier* on a memory or register reference indicates access to a single (4-byte) integer. Fields for the various `insn` annotations are not shown. ■

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.

```
(insn 8 6 10 (set (reg:SI 2)
        (mem:SI (symbol_ref:SI ("a")))))

(insn 10 8 12 (set (reg:SI 3)
        (mem:SI (symbol_ref:SI ("b")))))

(insn 12 10 14 (set (reg:SI 2)
        (plus:SI (reg:SI 2)
            (reg:SI 3))))

(insn 14 12 15 (set (reg:SI 3)
        (mem:SI (symbol_ref:SI ("c")))))

(insn 15 14 17 (set (reg:SI 2)
        (mult:SI (reg:SI 2)
            (reg:SI 3))))

(insn 17 15 19 (set (mem:SI (symbol_ref:SI ("d")))
        (reg:SI 2)))
```

Figure 14.14 Simplified textual version of the RTL for `d := (a + b) * c`.

CHECK YOUR UNDERSTANDING

24. Characterize Diana, GIMPLE, RTL, Java byte code, and Common Intermediate Language as high-, medium-, or low-level intermediate forms.
25. What is an *interface description language*?
26. Give a brief description of Diana.
27. Explain the distinction between *attributes* and *nodes* in Diana.
28. Name three languages (other than C) for which there exist `gcc` front ends.
29. What is the internal IF of `gcc`'s front ends?
30. Give brief descriptions of GIMPLE and RTL. How do they differ? Why was GIMPLE introduced?

14 Building a Runnable Program

14.7 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.

14.7.1 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 an entry point to a shared library can compute an appropriate base register value using the PC. Otherwise the caller must set the base register as part of the 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 14.22
PIC under MIPS/IRIX

Exact details vary among processors, vendors, and operating systems. Conventions for SGI's compilers for the MIPS architecture, under the IRIX 6.2 version of Unix, are illustrated in Figure 14.15. Each shared code segment is accompanied, at a static offset, by a nonshared *linkage table* and, at an arbitrary offset, by a nonshared data segment. The linkage table lists the addresses of all external symbols referenced in the code segment. Under MIPS/IRIX conventions, register gp (the "global pointer") is used to hold a reference to the linkage table.

As described in Section 8.2.2, any nonleaf subroutine must allocate space in its stack frame to hold the value of the `ra` (return address) register, and must save and restore this register in its prologue and epilogue. Similarly, any subroutine that may call into a dynamically linked shared library must save the gp register in the prologue, and restore it after every call into a different dynamically linked shared library. At code-generation time, the compiler must know which external symbols lie in such libraries. For a call to one of them, the usual `jal` (jump-and-link) instruction is replaced by a sequence of three instructions. The first of these loads register `t9` from the linkage table, using gp-relative addressing. The second is a `jalr` (jump-and-link-register) instruction, which takes its target address from `t9`. The third (to be executed after the return) restores the gp. In a similar vein, any load or store of a datum located in a dynamically linked shared library must employ a two-instruction sequence. The first instruction loads the address of the datum from the linkage table using gp-relative addressing. The second loads or stores the datum itself.

The prologue of any subroutine foo that serves as an entry to a dynamically linked shared library must establish a new gp. To do so it takes the value in `t9` (i.e., the address of foo) and adds the (statically known) signed distance between the code and the linkage table. ■

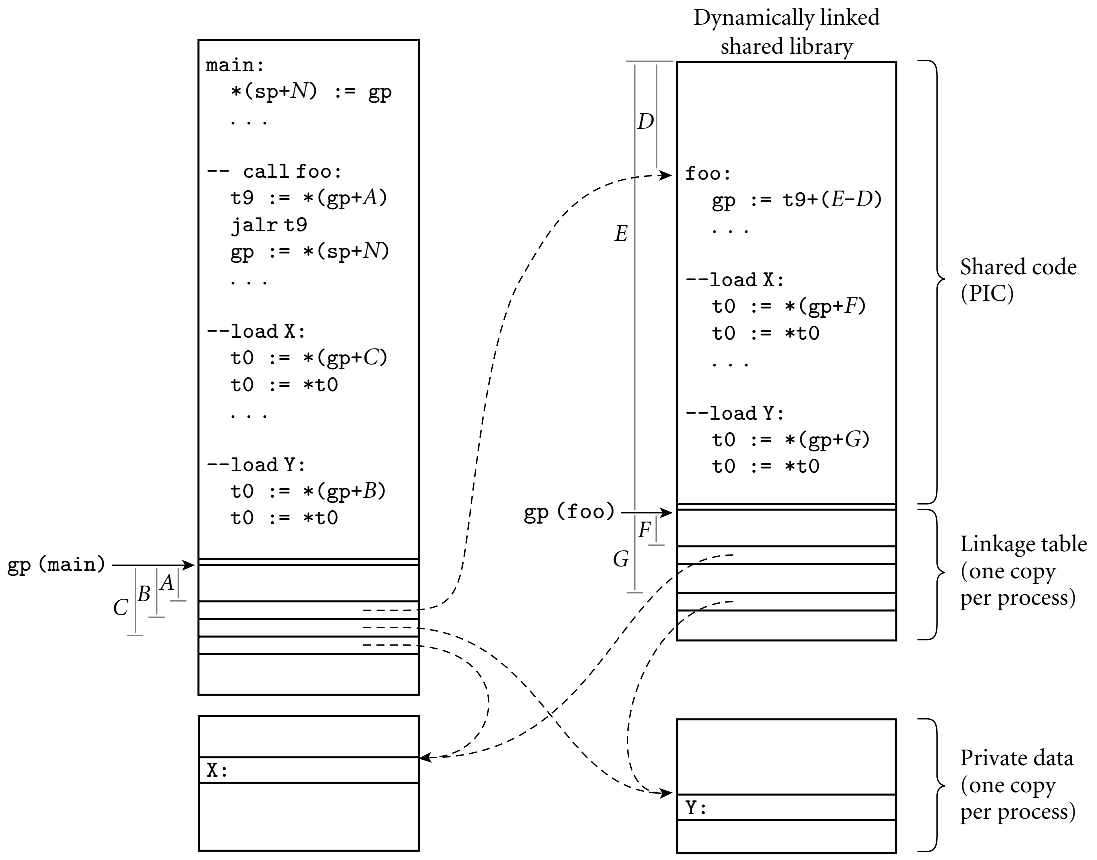


Figure 14.15 **A dynamically linked shared library.** Because `main` calls `foo`, which lies in the library, its prologue and epilogue must save and restore both `ra` (not shown) and `gp`. Calls to `foo` are made indirectly, using an address stored in `main`'s linkage table. Similarly, references to variables `X` and `Y`, both of which are globally visible, must employ a level of indirection. In the prologue of `foo`, `gp` is set to point to `foo`'s linkage table, using the value in `t9`. The calling sequence in `main` restores the old `gp` when `foo` returns.

14.7.2 Fully Dynamic (Lazy) Linking

If all or most of the symbols exported by a shared library are referenced by the parent program, then it makes sense to link the library in its entirety at load time. 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. Finally, a language system that allows the dynamic creation of program components (e.g., as in Common Lisp or Java) must use lazy linking to delay the resolution of external references in compiled components.

EXAMPLE 14.23
Dynamic linking under MIPS/IRIX

The run-time data structures for lazy linking are almost the same as those in Figure ©14.15, but they are incrementally created. At load time, the program begins with the main code segment and linkage table, and with all data segments for which addresses need to appear in that linkage table. In our specific example, we would load the data segments of both `main` and `foo`, because the addresses of both `X` (which belongs to `main`) and `Y` (which belongs to `foo`) need to appear in the main linkage table. We would not, however, load the code segment or linkage table of `foo`, despite the fact that the address of `foo` needs to appear in the linkage table. Instead, we would initialize that linkage table entry to refer to a *stub* routine, created by the compiler and included in the main code segment. The code of the stub looks like this:

```
t9 := *(gp+k)   -- lazy linker entry point
t7 := ra
t8 := n         -- index of stub
call *t9        -- overwrites ra
```

The lazy linker itself resides in a (nonlazy) shared library, linked to the program at load time. (Here we have assumed that its address lies at offset k in the linkage table.)

After branching to the lazy linker, control never returns to the stub. Instead, the linker uses the constant n to index into the import table of the program's object file, where it finds the information it needs to identify both the name and the library of the unresolved reference. The linker then loads the library's code segment into memory if it is not already there. At this point it can change ("patch") the linkage table entry through which the stub was called, so that it now points to the library routine. If it needed to load the library's code segment, the linker also creates a copy of the library's linkage table. It initializes all data entries in that table, loading (copies of) the segments to which those entries refer if they (the segments) have not already been loaded as part of an earlier linking operation. For each subroutine entry in the library's linkage table, the linker checks to see whether the relevant code segment has already been loaded. If so, it initializes the entry with the subroutine's address. If not, it initializes it with the address of its stub. Finally, the linker copies `t7` into `ra` and jumps to the newly linked library routine. At this point, everything appears as though the call had happened in the normal fashion. ■

As execution proceeds, further references to not-yet-loaded symbols extend the "frontier" of the program. Because invocations of the linker occur on subroutine calls and not on data references, the current frontier always includes a set of code segments and the data segments to which those code segments refer. Each linking operation brings in one new code segment, together with all of the additional data segments to which that code refers. If we were willing to intercept page faults, we could arrange to enter the linker on references to not-yet-loaded data. This approach would avoid loading data segments that are never really used, but the overhead of the faults might greatly increase execution time.

✓ CHECK YOUR UNDERSTANDING

31. Explain the addressing challenge faced by dynamic linking systems.
32. What is *position-independent code*? What is it good for? What special precautions must a compiler follow in order to produce it?
33. Under MIPS/IRIX conventions, explain the significance of the gp (global pointer) register in a program with dynamic linking.
34. What is the purpose of a *linkage table*?
35. What is *lazy* dynamic linking? What is its purpose? How does it work?

14 Building a Runnable Program

14.9 Exercises

14.12 Compare and contrast Diana and GIMPLE with each other and with the notation we have been using for syntax tree attribute grammars (Section 4.6).

14.13 **(a)** PC-relative branches on the MIPS are limited to an offset of $\pm 2^{17}$ bytes with respect to the current instruction. Explain how to generate position-independent code that needs to branch farther than this.

(b) Displacement on the MIPS is limited to an offset of $\pm 2^{15}$ bytes with respect to the specified base pointer. Explain how a dynamic library in the style of Figure ◎14.15 can access more than 2048 symbols.

14.14 In Example ◎8.66 we described how the Gnu Pascal compiler 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 ◎14.15, if the MIPS allowed instructions to be fetched from writable memory.

14 Building a Runnable Program

14.10 Explorations

14.20 Assuming you have access to `gcc`, run it with various of the compile-time flags that cause it to dump its RTL intermediate code. Recent versions of the compiler support about thirty such flags. Most have both a long descriptive name (e.g., `-fdump-rtl-cse` for a dump after common subexpression elimination) and a shorter abbreviation of the form `-d`**X**, where **X** is a single letter. Ask a local Unix guru to help you find and access the `gcc.info` files, which document RTL, the compile-time flags, and the various compiler phases.

14.21 Find out how linking works under your favorite operating system. Can code be dynamically linked? Can (nonprivileged) users create shared libraries? How does the loader find the libraries that need to be linked to a program? If your compiler can be instructed to generate position-independent code, how does this code compare (in size and run-time efficiency) with the non-position-independent equivalent?

14.22 Learn about *pointer swizzling* [Wil92a], originally developed to run programs on machines with insufficient virtual address space. Explain its connection to dynamic linking.

16 Code Improvement

In Chapter 14 we discussed the generation, assembly, and linking of target code in the 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 under a particular accounting system, 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 14, 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 14.1.1. 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 level of aggressiveness, many recent compilers 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 Ⓒ16.3) 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., minimization of register usage within 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 Ⓒ16.1 with a more detailed consideration of the phases of code improvement. We then turn to peephole optimization in Section Ⓒ16.2. It can be performed in the absence of other optimizations if desired, and the discussion introduces some useful terminology. In Sections Ⓒ16.3 and Ⓒ16.4 we consider local and global redundancy elimination. Sections Ⓒ16.5 and Ⓒ16.7 cover code improvement for loops. Section Ⓒ16.6 covers instruction scheduling. Section Ⓒ16.8 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; "intraprocedural" might be more accurate.

16.1 Phases of Code Improvement

As we noted in Chapter 14, the structure of the back end varies considerably from compiler to compiler. For simplicity of presentation we will continue to focus on the structure introduced in Section 14.1. 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.

EXAMPLE 16.1
Code improvement phases

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 ◎16.1. Within this structure, the machine-independent part of the back 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 part of the 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 ◎16.1 is shown as three separate 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 ◎16.4, ◎16.5, and ◎16.7, 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 ◎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

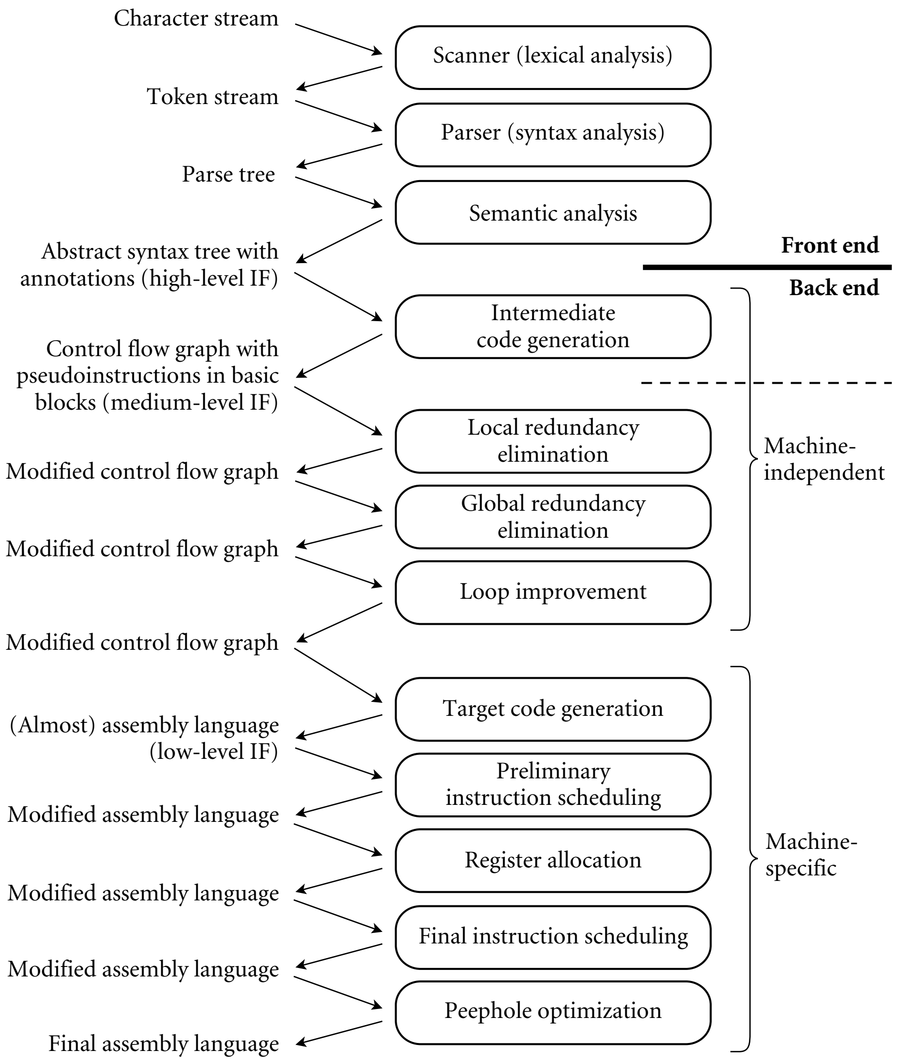

Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis
Abstract syntax tree with annotations (high-level IF)
Front end
Back end
Intermediate code generation
Control flow graph with pseudoinstructions in basic blocks (medium-level IF)
Local redundancy elimination
Modified control flow graph
Global redundancy elimination
Modified control flow graph
Loop improvement
Machine-independent
Modified control flow graph
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 16.1 **A more detailed view of the compiler structure originally presented in Figure 14.1 (page 731).** 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.

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* registers temporarily to memory. The second round of instruction scheduling attempts to fill any delays induced by the extra loads.

16.2 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 14, however, the result is generally of very poor quality (contrast the code of Example 1.2 (page 5) with that of Figure 1.6). 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 16.2

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 16.3

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 16.4

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 + r2  becomes    r3 := r1 + 4   and then   r3 := r1 + 4
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 + r2  becomes    r3 := r1 + 4   and then   r3 := *(r1 +4)
r3 := *r3                 r3 := *r3
```

(This assumes again that r2 is dead.)

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 16.5

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 16.6

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 16.7

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 16.1.) In a similar vein, algebraic identities allow us to perform simplifications like the following:

```
r1 := r2 × 0    becomes    r1 := 0
```

■

EXAMPLE 16.8

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 5.5.1.

EXAMPLE 16.9

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 the list above 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, 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.

DESIGN & IMPLEMENTATION

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.

16.3 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 14.1.1. 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 14.6 (page 740) we presented an attribute grammar to generate linear (goto-containing) code for simple syntax trees. A similar grammar can be used to create a control flow graph (Exercise 14.6).

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 8.2.4, 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.

16.3.1 A Running Example

EXAMPLE 16.10
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 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:

```
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 ◎16.2) that the use of integer arithmetic will not lead to round-off errors. ■

EXAMPLE 16.11
Syntax tree and naive control flow graph

A syntax tree for our subroutine appears in Figure ◎16.2, with basic blocks identified. The corresponding control flow graph appears in Figure ◎16.3.

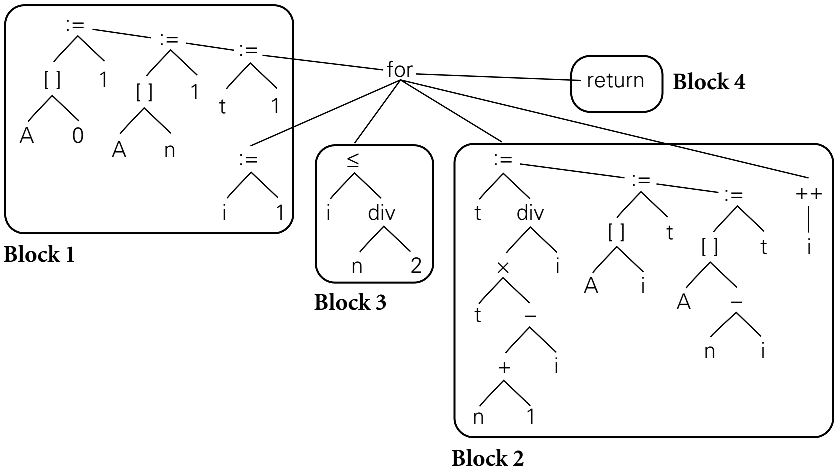


Figure 16.2 Syntax tree for the `combinations` subroutine. Portions of the tree corresponding to basic blocks have been circled.

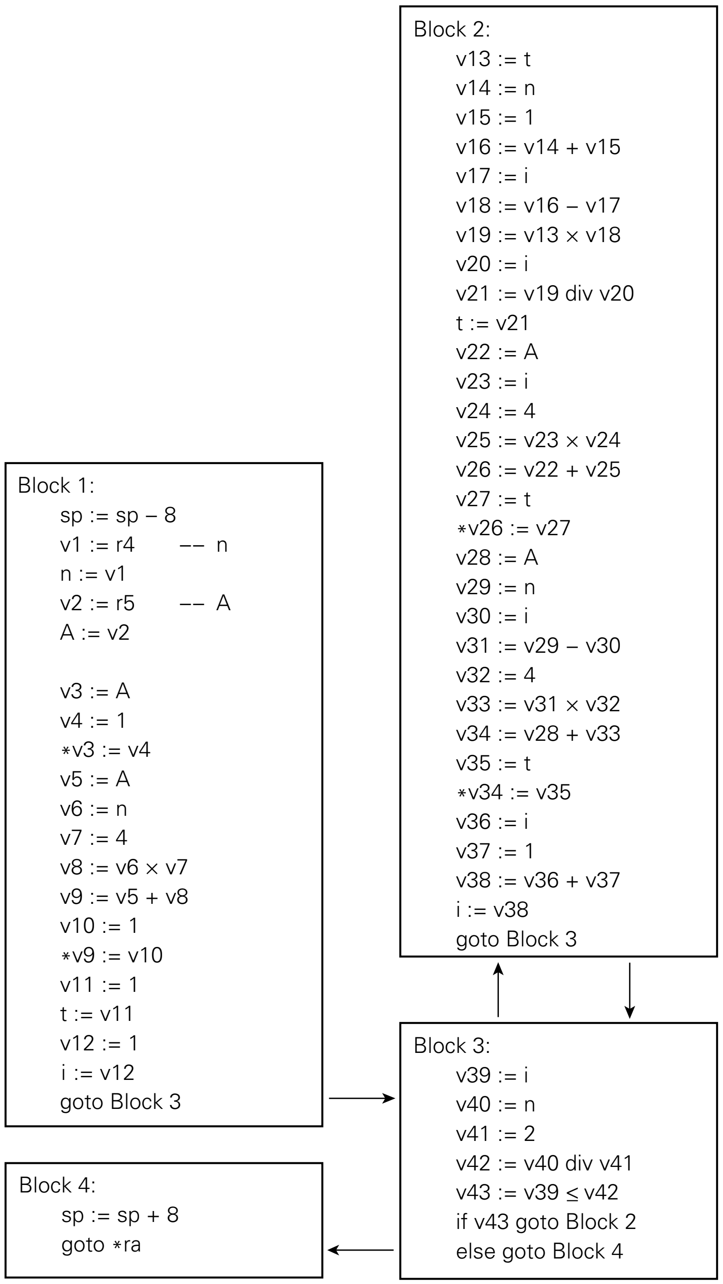
To avoid artificial interference between instructions at this early stage of code improvement, we employ a medium-level intermediate form (IF) in which every calculated value is placed in a separate register. To emphasize that these are virtual registers (of which there is an unlimited supply), we name them v1, v2, ... We will use r1, r2, . . . to represent architectural registers in Section ◎16.8.

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

DESIGN & IMPLEMENTATION

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 ◎16.4.2, ◎16.5, and ◎16.7, 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.



```
Block 1:
      sp := sp − 8
      v1 := r4       -- n
      n := v1
      v2 := r5       -- 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 *ra
```

Figure 16.3 **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.

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 the MIPS calling conventions, described in Section ©8.2.2. 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 (ra) or frame pointer (fp) registers. In all cases, references 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 r4 and r5 on the MIPS), our original (naive) code immediately saves these values to memory, so that references 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 ©16.4.1. 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. ■

16.3.2 **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; FL88, Sec. 15.7]. 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. 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 our 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.

DESIGN & IMPLEMENTATION

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 7.4.3), references to variables in lexically enclosing scopes (Section 8.2), and references to nearby fields in complex records (Section 7.3.2). Like the pointer arithmetic discussed in the sidebar on page 354, hand elimination of common subexpressions, unless it makes the code easier to read, is usually not a good idea.

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 +c̃`; 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 the sidebar on page ◎334).

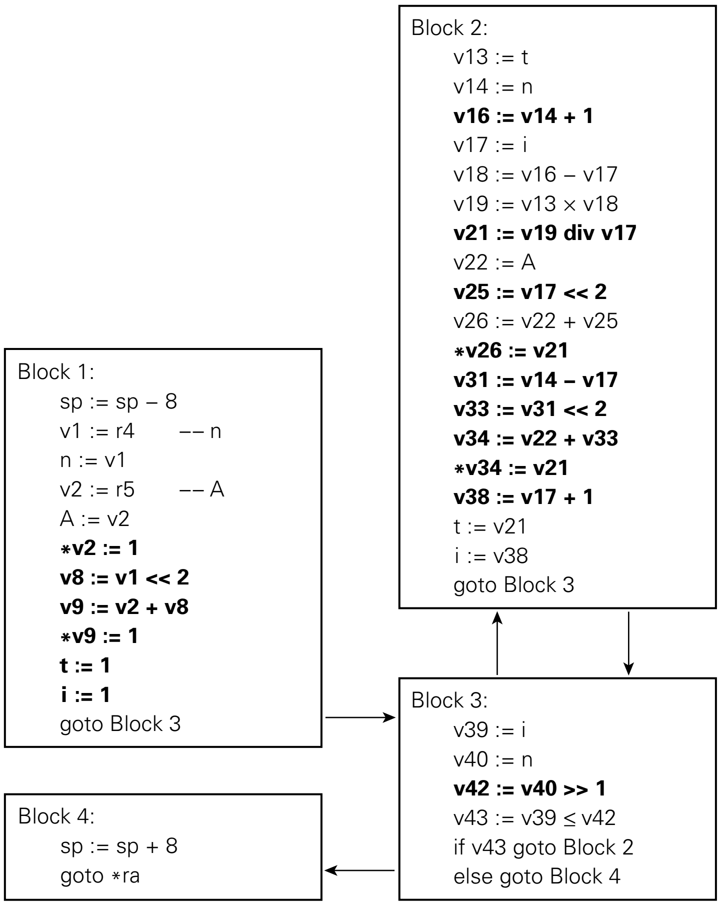
EXAMPLE 16.12
Result of local redundancy elimination

Figure ◎16.4 shows the control flow graph for our `combinations` subroutine after local redundancy elimination. We have eliminated 21 of the instructions in Figure ◎16.3, all of them loads of variables or constants. Thirteen of the eliminated 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. ■

DESIGN & IMPLEMENTATION

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. Until recently Fortran had no pointers, and many Fortran programs are still written without them. C programs, by contrast, tend to be pointer-rich. Alias analysis for pointers is an active research topic, and has reached the point at which good C compilers can often rival their Fortran counterparts. For a survey of the state of the art (as of the turn of the century), see the paper by Michael Hind [Hin01].



```
Block 1:
      sp := sp − 8
      v1 := r4       -- n
      n := v1
      v2 := r5       -- 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 *ra
```

Figure 16.4 **Control flow graph for the `combinations` subroutine after local redundancy elimination and strength reduction.** Changes from Figure 16.3 are shown in boldface type.

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?

16.4 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 ©16.5 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.

16.4.1 SSA Form and Global Value Numbering

Value numbering, as introduced in Section ©16.3, 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.

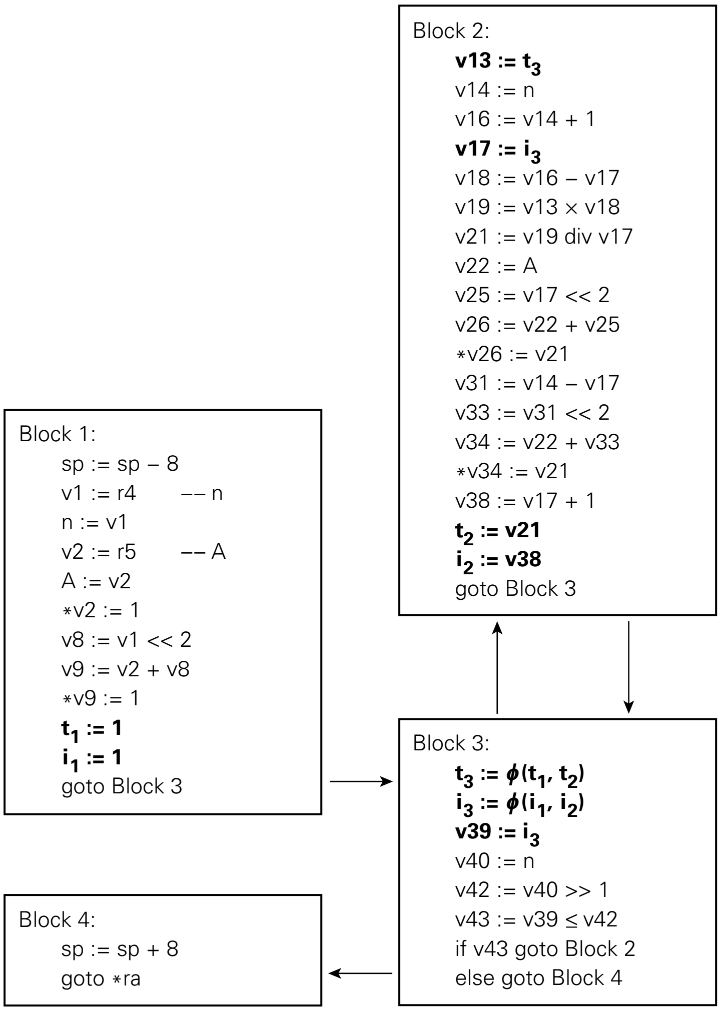
Following the flow of the program, we assign subscripts to variables in load instructions, to match the corresponding stores. If the instruction `v2 := x` is guaranteed to read the value of `x` written by the instruction x_3 `:= v1`, then we replace `v2 := x` with `v2 :=` x_3. If we cannot tell which 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 16.4.2. In the current subsection we will generate SSA code informally; data flow formulations can be found in more advanced compiler texts [CT04, Sec. 9.3; AK02, Sec. 4.4.4; App97, Sec. 19.1; Muc97, Sec. 8.11].

EXAMPLE 16.13
Conversion to SSA form

In the `combinations` subroutine (Figure 16.4) 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 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 16.5. ■



```
Block 1:
    sp := sp − 8
    v1 := r4        −− n
    n := v1
    v2 := r5        −− 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 ∗ra
```

Figure 16.5 Control flow graph for the `combinations` subroutine, in static single assignment (SSA) form. Changes from Figure 16.4 are shown in boldface type.

EXAMPLE 16.14
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. Again, we perform the analysis for our running example informally.

We can begin by adopting the virtual register names in Block 1; since local redundancies have been removed, these 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.

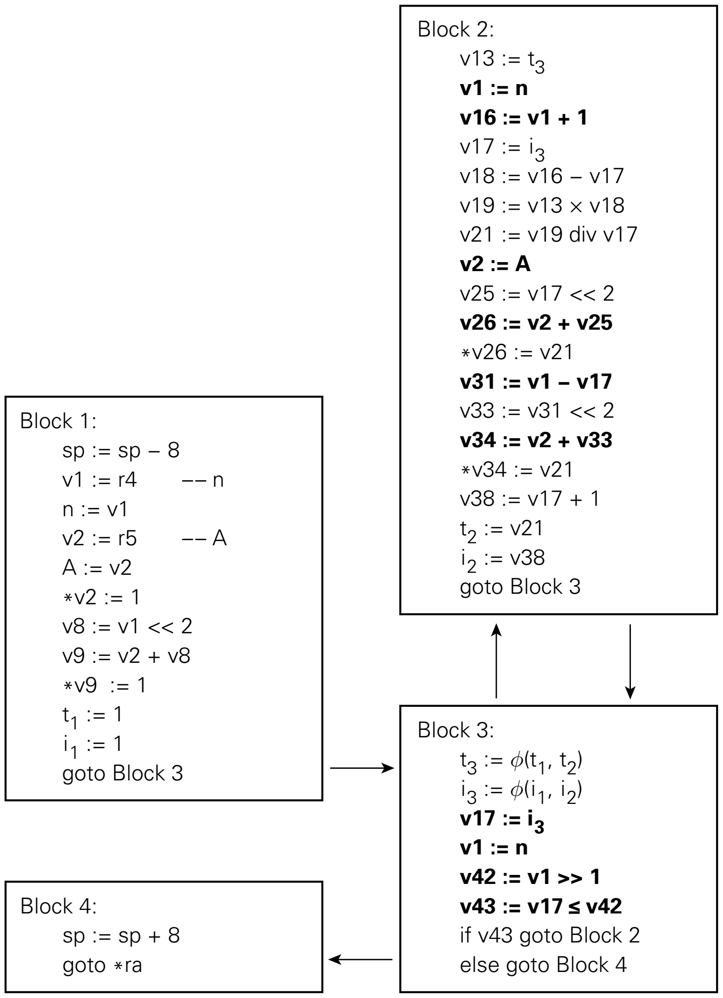
The result of global value numbering on our `combinations` subroutine appears in Figure 16.6. In this case the only common values identified were variables loaded from memory. In a more complicated subroutine, we would also identify known-to-be-identical computations performed in more than one block (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.

16.4.2 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 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

[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.



```
Block 1:
    sp := sp − 8
    v1 := r4      −− n
    n := v1
    v2 := r5      −− 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 *ra
```

Figure 16.6 **Control flow graph for the `combinations` subroutine after global value numbering.** Changes from Figure 16.5 are shown in boldface type.

common subexpression elimination is to identify places in which an instruction that computes 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 16.15
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]

$$Out_B = Gen_B \cup (In_B \smallsetminus Kill_B)$$
$$In_B = \bigcap_{\text{predecessors } A \text{ of } B} Out_A$$

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 16.16
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 2.3.2. 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 conservatively 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. 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

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.

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:

$$\begin{array}{ll}
Gen_1 = \{\text{v1}, \text{v2}, \text{v8}, \text{v9}\} & Kill_1 = \{\text{v13}, \text{v16}, \text{v17}, \text{v26}, \text{v31}, \text{v34}, \text{v42}\} \\
Gen_2 = \{\text{v1}, \text{v2}, \text{v13}, \text{v16}, \text{v17}, \text{v18}, \text{v19}, & Kill_2 = \{\text{v8}, \text{v9}, \text{v13}, \text{v17}, \text{v42}, \text{v43}\} \\
\quad \text{v21}, \text{v25}, \text{v26}, \text{v31}, \text{v33}, \text{v34}, \text{v38}\} & \\
Gen_3 = \{\text{v1}, \text{v17}, \text{v42}, \text{v43}\} & Kill_3 = \{\text{v8}, \text{v16}, \text{v18}, \text{v21}, \text{v25}, \text{v31}, \text{v38}\} \\
Gen_4 = \varnothing & Kill_4 = \varnothing
\end{array}$$

Applying the first of our data flow equations ($Out_B = Gen_B \cup (In_B \setminus Kill_B)$) to all blocks, we obtain

$$\begin{array}{l}
Out_1 = \{\text{v1}, \text{v2}, \text{v8}, \text{v9}\} \\
Out_2 = \{\text{v1}, \text{v2}, \text{v13}, \text{v16}, \text{v17}, \text{v18}, \text{v19}, \text{v21}, \text{v25}, \text{v26}, \text{v31}, \text{v33}, \text{v34}, \text{v38}\} \\
Out_3 = \{\text{v1}, \text{v2}, \text{v9}, \text{v13}, \text{v17}, \text{v19}, \text{v26}, \text{v33}, \text{v34}, \text{v42}, \text{v43}\} \\
Out_4 = \{\text{v1}, \text{v2}, \text{v8}, \text{v9}, \text{v13}, \text{v16}, \text{v17}, \text{v18}, \text{v19}, \text{v21}, \text{v25}, \text{v26}, \text{v31}, \text{v33}, \text{v34}, \text{v38}, \text{v42}, \text{v43}\}
\end{array}$$

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

$$\begin{array}{ll}
In_1 = \varnothing & Out_1 = \{\text{v1}, \text{v2}, \text{v8}, \text{v9}\} \\
In_2 = \{\text{v1}, \text{v2}, \text{v9}, \text{v13}, \text{v17}, \text{v19}, & Out_2 = \{\text{v1}, \text{v2}, \text{v13}, \text{v16}, \text{v17}, \text{v18}, \text{v19}, \\
\quad \text{v26}, \text{v33}, \text{v34}, \text{v42}, \text{v43}\} & \quad \text{v21}, \text{v25}, \text{v26}, \text{v31}, \text{v33}, \text{v34}, \text{v38}\} \\
In_3 = \{\text{v1}, \text{v2}\} & Out_3 = \{\text{v1}, \text{v2}, \text{v17}, \text{v42}, \text{v43}\} \\
In_4 = \{\text{v1}, \text{v2}, \text{v9}, \text{v13}, \text{v17}, \text{v19}, & Out_4 = \{\text{v1}, \text{v2}, \text{v9}, \text{v13}, \text{v17}, \text{v19}, \\
\quad \text{v26}, \text{v33}, \text{v34}, \text{v42}, \text{v43}\} & \quad \text{v26}, \text{v33}, \text{v34}, \text{v42}, \text{v43}\}
\end{array}$$

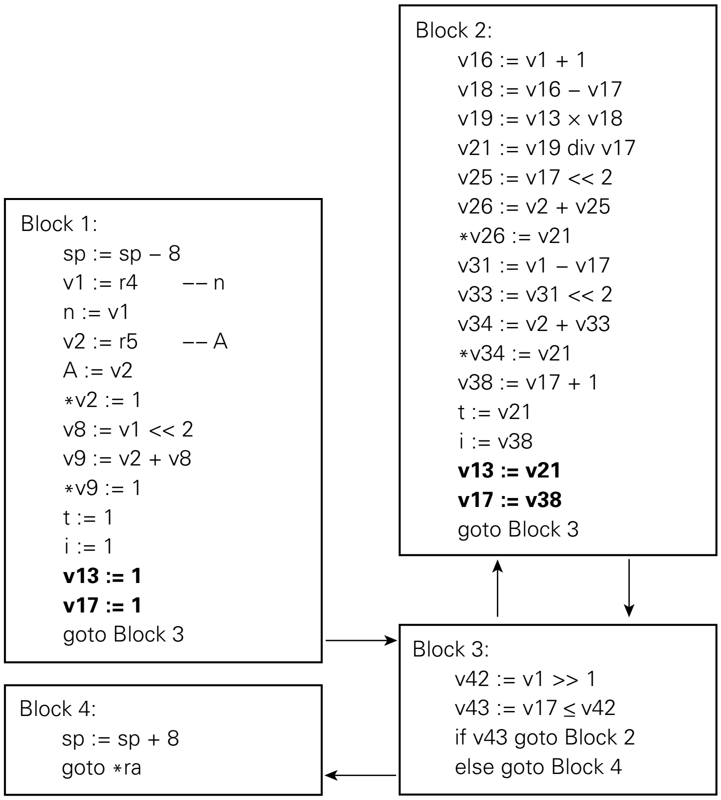
One more iteration of each equation yields the fixed point:

$$\begin{array}{ll}
In_1 = \varnothing & Out_1 = \{\text{v1}, \text{v2}, \text{v8}, \text{v9}\} \\
In_2 = \{\text{v1}, \text{v2}, \text{v17}, \text{v42}, \text{v43}\} & Out_2 = \{\text{v1}, \text{v2}, \text{v13}, \text{v16}, \text{v17}, \text{v18}, \text{v19}, \\
 & \quad \text{v21}, \text{v25}, \text{v26}, \text{v31}, \text{v33}, \text{v34}, \text{v38}\} \\
In_3 = \{\text{v1}, \text{v2}\} & Out_3 = \{\text{v1}, \text{v2}, \text{v17}, \text{v42}, \text{v43}\} \\
In_4 = \{\text{v1}, \text{v2}, \text{v17}, \text{v42}, \text{v43}\} & Out_4 = \{\text{v1}, \text{v2}, \text{v17}, \text{v42}, \text{v43}\}
\end{array}$$

EXAMPLE 16.17

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



```
Block 1:
    sp := sp − 8
    v1 := r4       -- n
    n := v1
    v2 := r5       -- 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 *ra
```

Figure 16.7 Control flow graph for the `combinations` subroutine after performing global common subexpression elimination. Note the absence of the many load instructions of Figure 16.6. Compensating register–register moves are shown in boldface type. Live variable analysis will allow us to drop the two pairs of instructions immediately before these moves, together with the stores of n and A (v1 and v2) in Block 1. It will also allow us to drop changes to the stack pointer in the subroutine prologue and epilogue: we won't need space for local variables anymore.

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 16.7.

(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 16.5.2 and 16.8, respectively.) ■

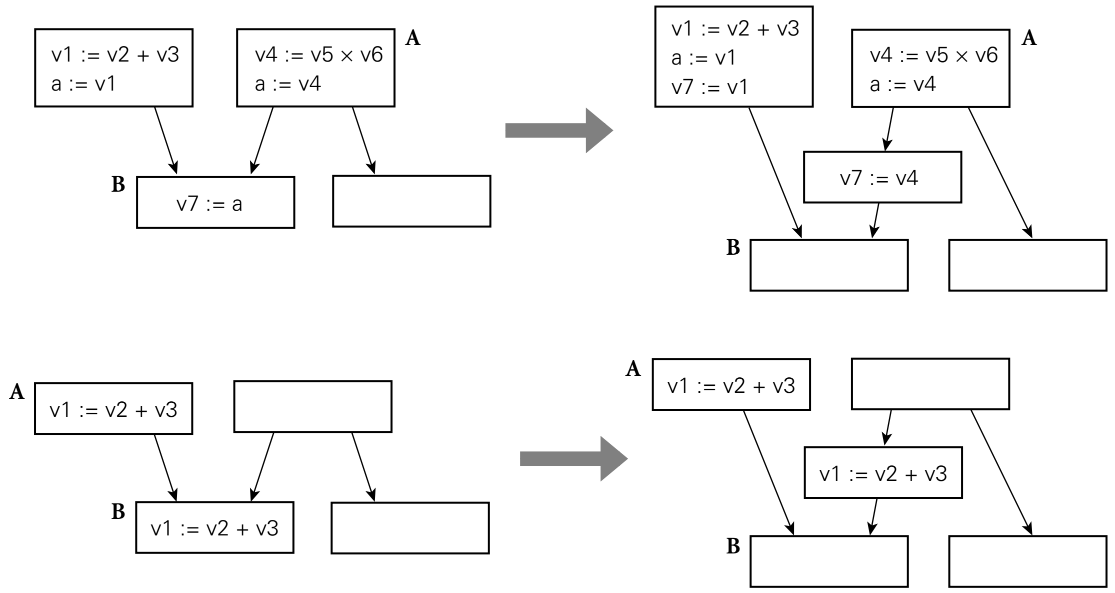


Figure 16.8 Splitting an edge of a control flow graph to eliminate a redundant load (*top*) or a partially redundant computation (*bottom*).

Splitting Control Flow Edges

EXAMPLE 16.18
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 16.8. 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 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 can help to identify other dead instructions. (None of these arise this early in the `combinations` example.)

EXAMPLE 16.19
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:

$$In_B = Gen_B \cup (Out_B \smallsetminus Kill_B)$$
$$Out_B = \bigcup_{\text{successors } C \text{ of } B} In_C$$

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 ©16.7 and ©16.9. ■

EXAMPLE 16.20
Fixed point for live variables

In our example program, we have:

$$\begin{aligned} Gen_1 &= \varnothing & \quad Kill_1 &= \{\mathsf{n}, \mathsf{A}, \mathsf{t}, \mathsf{i}\} \\ Gen_2 &= \varnothing & Kill_2 &= \{\mathsf{t}, \mathsf{i}\} \\ Gen_3 &= \varnothing & Kill_3 &= \varnothing \\ Gen_4 &= \varnothing & Kill_4 &= \varnothing \end{aligned}$$

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 are live; all of the stores of `A`, `n`, `t`, and `i` can be eliminated (see Figure ◎16.7). In addition, the fact that `t` and `i` can be kept entirely in registers means that we won't need to update the stack pointer in the subroutine prologue and epilogue: there won't be any stack frame. ■

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.

16.5 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 ◎16.7 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.

16.5.1 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 (e.g., Ada or Java): 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 16.21
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 ◎16.4.1, 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

$$Out_B = Gen_B \cup (In_B \smallsetminus Kill_B)$$
$$In_B = \bigcup_{\text{predecessors } C \text{ of } B} Out_C$$

DESIGN & IMPLEMENTATION

Loop invariants

Many loop invariants arise from address calculations, especially for arrays. Like the common subexpressions discussed in the sidebar on page ◎333, 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 (page 178).

Our initial condition is that $In_1 = \varnothing$: 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 $In_B = \varnothing$ for all blocks B (not just B_1). ■

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 16.22
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 ©16.7) yields the code of Figure ©16.9. ■

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 ©16.4. In the specific case of the `combinations` subroutine, our more naive transformation is both safe and (in the common case) efficient.

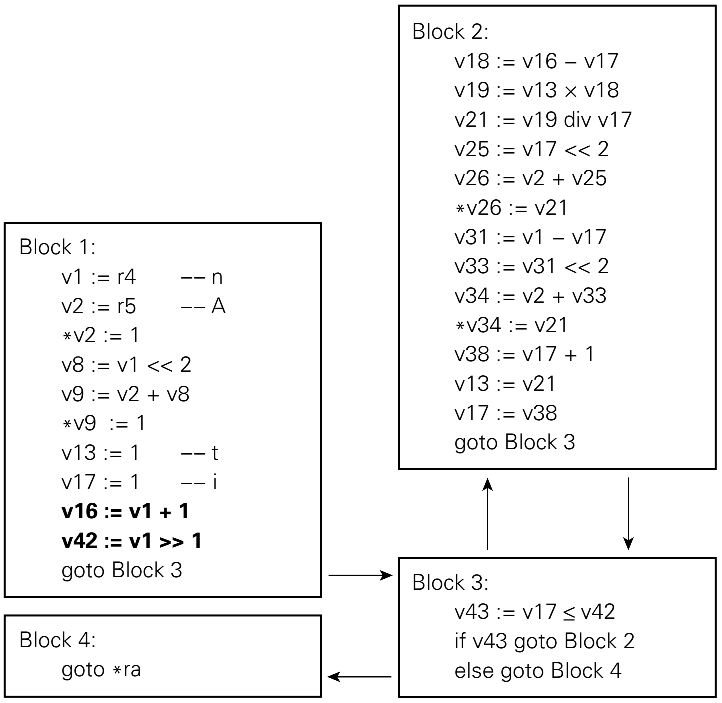
16.5.2 Induction Variables

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

DESIGN & IMPLEMENTATION

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 := r4       -- n
    v2 := r5       -- 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 *ra
```

Figure 16.9 **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 ◎16.7, and have eliminated the stack space for t and i, which now reside entirely in registers.

arithmetic progressions; more elaborate examples appear in Exercises ◎16.11 and ◎16.12. 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 16.23
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, 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 16.24
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 ◎16.10 for an example). ■

```
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)

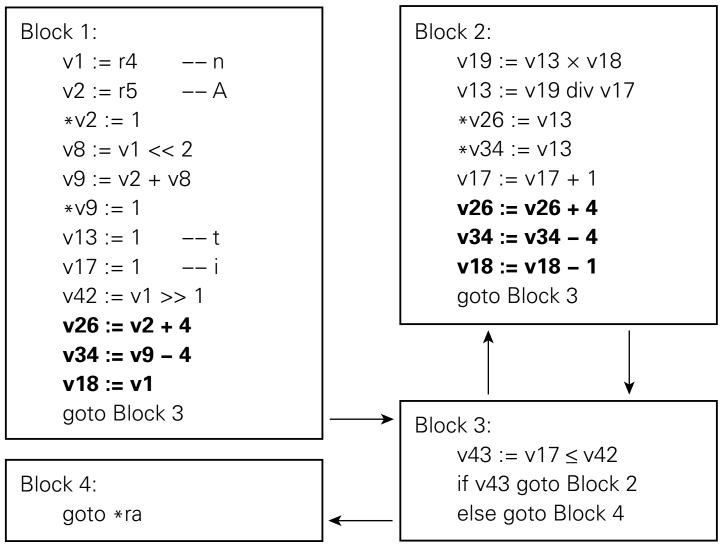
Figure 16.10 **Code improvement of induction variables.** High-level pseudocode source is shown in (a). Target code prior to induction variable optimizations is shown in (b). In (c) we have performed strength reduction on v5, the array index, and eliminated v4, at which point v5 no longer depends on v1 (i). In (d) we have modified the end test to use v5 instead of v1, and have eliminated v1.

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.

EXAMPLE 16.25
Result of induction variable optimization

For our `combinations` example, the code resulting from induction variable optimizations appears in Figure 16.11. 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.



```
Block 1:
    v1 := r4        -- n
    v2 := r5        -- 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 *ra
```

Figure 16.11 **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.

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 redundant instruction in the block, and allows us to discuss instruction scheduling in comparative isolation from other issues. ■

16.6 Instruction Scheduling

In the example compiler structure of Figure 16.1, the next phase after loop optimization is target code generation. As noted in Chapter 14, 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-assembler "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-assembler, we won't consider peephole optimization in any further detail. In Section 16.7, 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, performance depends critically on the extent to which the compiler is able to keep the pipeline full. As explained in Section 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 16.26
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 16.3 into the eight-instruction sequence of Figure 16.11 (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 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 an increasing number of machines by hardware register renaming.

EXAMPLE 16.27
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

5 What we are discussing here is a *dependence DAG*. It is related to, but distinct from, the expression DAG mentioned in Section 16.3. 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
```

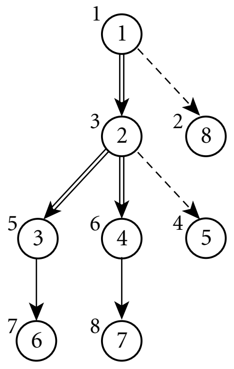


Figure 16.12 **Dependence DAG for Block 2 of Figure 16.11, together with pseudocode for the entire loop, both before (left) and after (right) instruction scheduling.** Circled numbers in the DAG correspond to instructions in the original version of the loop. Smaller adjacent numbers give the schedule 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 16.20), only two instructions can be moved.

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. If our target machine had condition codes (described in Section 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 16.12. 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 16.28
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 16.7 we will consider other versions of the block, in which rescheduling yields significantly faster code. ■

As noted near the end of Section 16.1, 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.

16.7 Loop Improvement II

As noted in Section 16.5, 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].

16.7.1 Loop Unrolling and Software Pipelining

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, and allowing the scheduler to

EXAMPLE 16.29
Result of loop unrolling

intermingle the instructions of the original iterations. If we unroll two iterations of our `combinations` example we obtain the code of Figure 16.13. 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 16.30
Result of software pipelining

If we unroll the loop three times instead of two (see Exercise 16.21), 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 16.14. 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 16.14, and as a control flow graph in Figure 16.15. 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

```
Block 1:
        ...             -- code from Block 1, figure 15.11
        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
        v22 := v22 − 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 := v19 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              —
        —                             —
        —                             —
        —                             —
        —                             *(v36+4) := v13
 10.    v13 := t19 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 4
        -- else fall through to Block 4
```

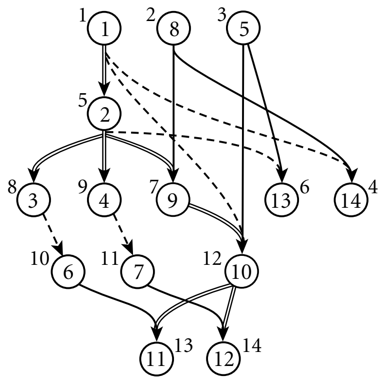


Figure 16.13 Dependence DAG for Block 2 of the `combinations` subroutine after unrolling two iterations of the body of the loop. Also shown is linearized pseudocode for the entire loop, both before (left) and after (right) instruction scheduling. New instructions added to the end of Block 1 cover the case in which the number of iterations of the original loop is not a multiple of two.

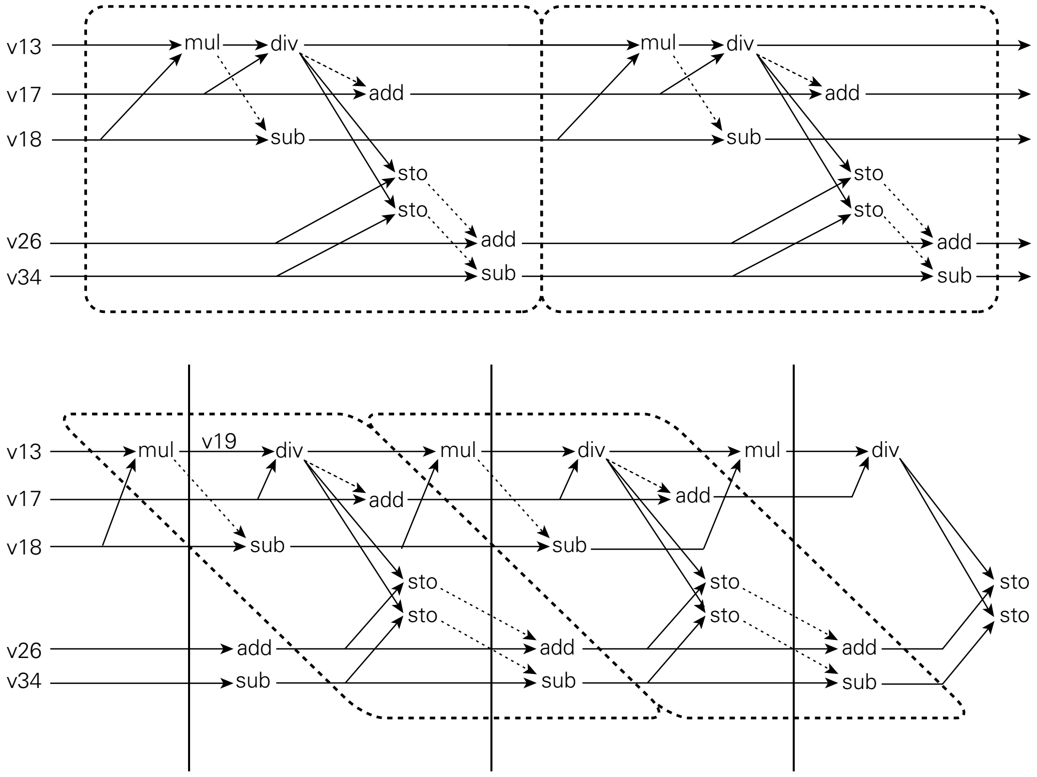
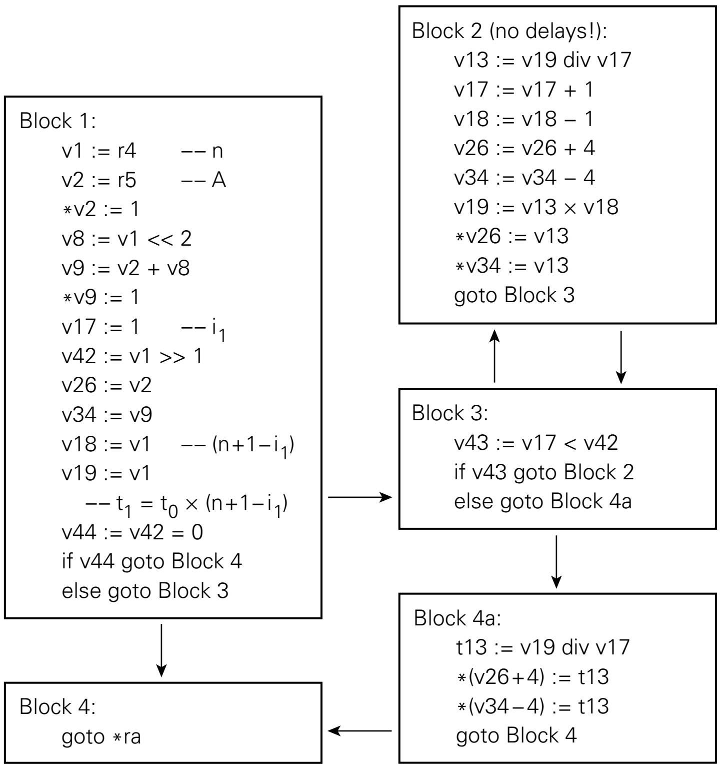


Figure 16.14 **Software pipelining.** The top diagram illustrates the execution of the original (nonpipelined) loop. In the bottom diagram, each iteration of the original loop has been spread across three iterations of the pipelined loop. Iterations of the original loop are enclosed in a dashed-line 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.

identity: whereas the original loop carried the result of the divide around to the next multiply 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 ten. 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 := r4        -- n
    v2 := r5        -- 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 *ra
```

Figure 16.15 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 $\leq$), 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 16.22).

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.

16.7.2 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 in recent years. First, as improvements in processor speed have continued to outstrip 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 16.31
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 16.32
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]
```

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 ©16.25).

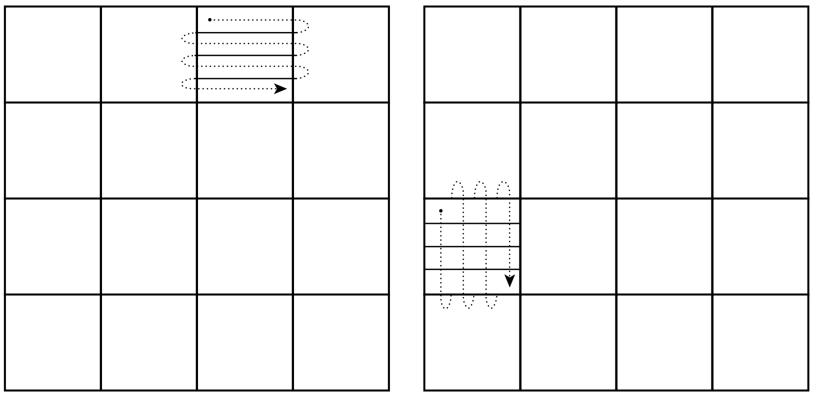


Figure 16.16 **Tiling (blocking) of a matrix operation.** As long as one tile of A and one tile of B can fit in the cache simultaneously, only one access in m will cause a cache miss (where m is the number of elements per cache line).

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 ◎16.16. 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 ◎16.23 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 16.33
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]];
```

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]];
```
■

6 Although B is being written, not read, the hardware will fetch each line of B 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.

EXAMPLE 16.34
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 16.35
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

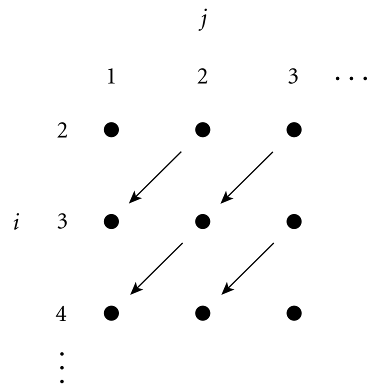
When reordering loops, we must be extremely careful to respect all data dependences. Of particular concern are so-called *loop-carried* dependences, which

EXAMPLE 16.36
Loop-carried dependences

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]. ■

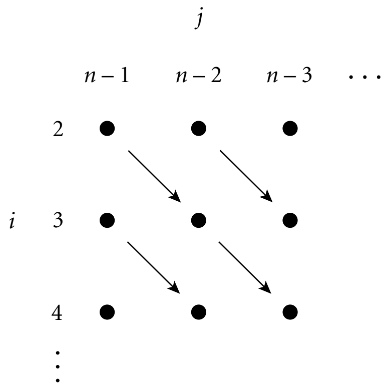
To analyze loop-carried dependences, high-performance optimizing compilers use symbolic mathematics to characterize the sets of index values that may cause the subscript expressions in different array references to evaluate to the same value. Compilers differ somewhat in the sophistication of this analysis. Most can handle linear combinations of loop indices. None, of course, can handle all expressions, since equivalence of general formulae is uncomputable. When unable to fully characterize subscripts, a compiler must conservatively assume the worst, and rule out transformations whose safety cannot be proven.

EXAMPLE 16.37
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 16.36 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:

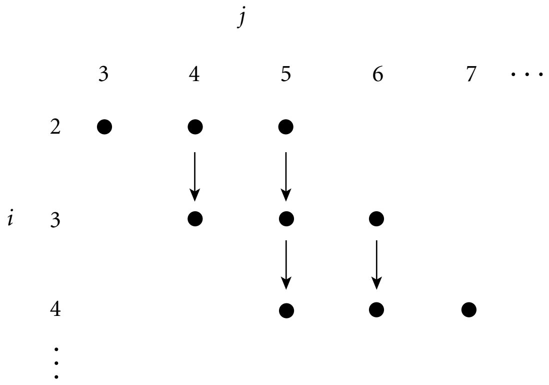
```
for j := n–1 to 1 by–1
    for i := 2 to n
        A[i, j] := A[i, j] – A[i–1, j+1]
```

EXAMPLE 16.38
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 machine or a multiprocessor. 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 12.10 and Section 12.4.5 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.

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 "zero-ing out" a two-dimensional array, here indexed from 0 to $n-1$ and laid out in row-major order:

EXAMPLE 16.39
Coarse-grain parallelization

```
for i := 0 to n–1
    for j := 0 to n–1
        A[i, j] := 0
```

On a machine containing several general-purpose processors, we would probably parallelize the outer loop:

```
– – on processor 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 processor a band of rows to initialize. We have assumed that processors 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 16.40
Strip mining

```
for i := 0 to n−1
    for j := 0 to n/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 uniprocessor, locality in parallel programs reduces communication among processors and between the processors and memory. Optimizations similar to those employed to reduce the number of cache misses on a uniprocessor can be used to reduce communication traffic on a multiprocessor.

Load balance refers to the division of labor among processors on a parallel machine. If we divide the work of a program among 16 processors, we shall obtain a speedup of close to 16 only if each processor takes the same amount of time to do its work. If we accidentally assign 5% of the work to each of 15 processors 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 processors 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 12.2. Each task consists of a set of loop iterations. The number of such tasks is chosen to be significantly larger than the number of processors. When finished with a given task, a processor goes back to the bag to get another.

16.8 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, the implicit pointers of `with` statements in Pascal-family languages (Section ◎7.3.3), 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] in the early 1980s, 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 [CT04, Chap. 13].

EXAMPLE 16.41
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 ◎16.5.1). For the software-pipelined version of our `combinations` subroutine (Figure ◎16.15, page ◎359), we can chart the *live ranges* of the virtual registers as shown in Figure ◎16.17. 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 16.42
Register coloring

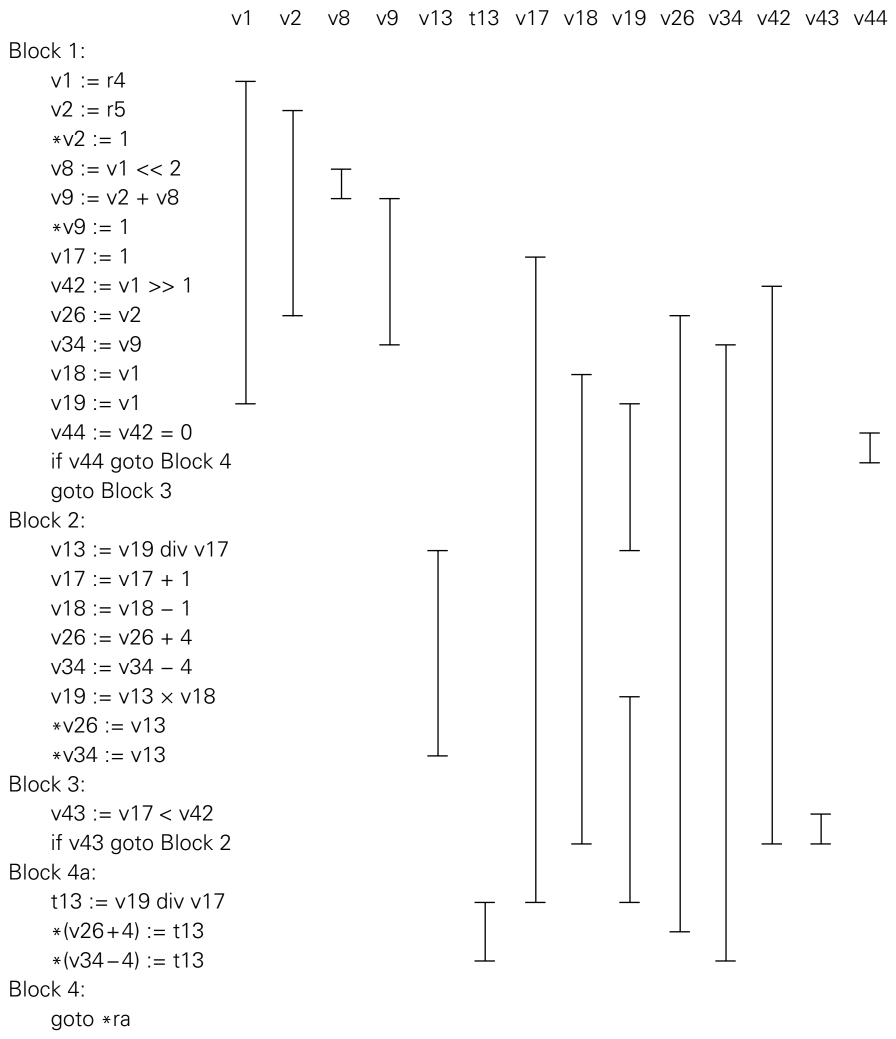
Given these live ranges, we construct a *register interference graph.* The nodes of this graph represent virtual registers. Registers v*i* and v*j* are connected by an arc if they are simultaneously live. The interference graph corresponding to Figure ◎16.17 appears in Figure ◎16.18. The problem of mapping 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 16.43
Optimized `combinations` subroutine

Final code for the `combinations` subroutine appears in Figure ◎16.19. We have left `v1`/`v19` and `v2`/`v26` in `r4` and `r5`, the registers in which their initial values were passed. Because our subroutine is a leaf, these registers are never needed for other arguments. Following MIPS conventions (Section ◎5.4.5), we have used registers `r8` through `r12` as additional temporary registers. ■

We have glossed over two important issues. First, on almost any real machine, architectural registers are not uniform. Integer registers cannot be used for



```
Block 1:
    v1 := r4
    v2 := r5
    *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 *ra
```

Figure 16.17 Live ranges for virtual registers in the software-pipelined version of the `combinations` subroutine (Figure 16.15).

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 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 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

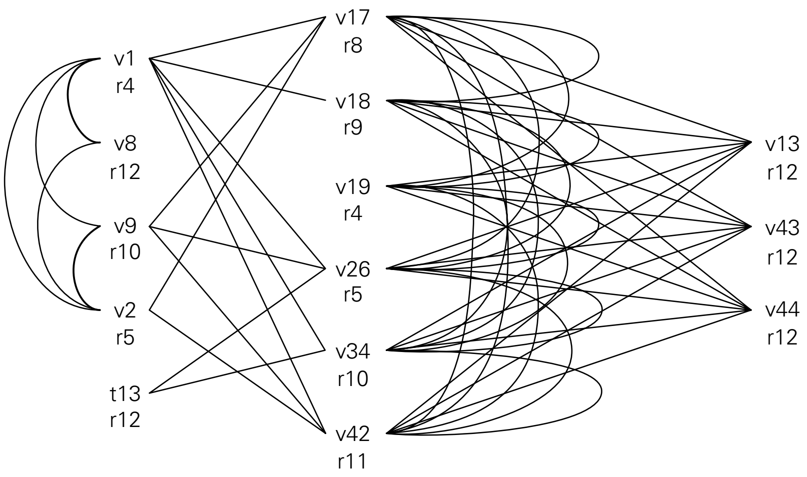


Figure 16.18 **Register interference graph for the software pipelined version of the `combinations` subroutine.** Using architectural register names, we have indicated one of several possible seven-colorings.

```
Block 1:
    ∗r5 := 1
    r12 := r4 << 2
    r10 := r5 + r12
    ∗r10 := 1
    r8 := 1
    r11 := r4 >> 1
    r9 := r4
    r12 := r11 = 0
    if r12 goto Block 4
    goto Block 3
Block 2:
    r12 := r4 div r8
    r8 := r8 + 1
    r9 := r9 − 1
    r5 := r5 + 4
    r10 := r10 − 4
    r4 := r12 × r9
    ∗r5 := r12
    ∗r10 := r12
Block 3:
    r12 := r8 < r11
    if r12 goto Block 2
Block 4a:
    r12 := r4 div r8
    ∗(r5+4) := r12
    ∗(r10−4) := r12
Block 4:
    goto ∗ra
```

Figure 16.19 **Final code for the `combinations` subroutine, after assigning architectural registers and eliminating useless copy instructions.**

colors. After coloring the resulting graph, we assign each virtual register to the architectural register of the same color.

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 Ⓒ16.1 and Ⓒ16.6, 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 multiprocessors 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 the architectural registers of a microprocessor uniformly for purposes of register allocation.

16.9 Summary and Concluding Remarks

This chapter has addressed the subject of code improvement ("optimization"). We considered many of the most important optimization techniques, including peephole optimization; local and global (intrasubroutine) 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, instruction scheduling and register allocation are definitely on the list: basic-block level scheduling and elimination of redundant loads and stores are crucial in any production-quality compiler. 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 increasingly interrelated.

16.10 Exercises

16.1 In Section ◎16.2 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.

16.2 Prove that the division operation in the loop of the `combinations` subroutine (Example ◎16.10) always produces a remainder of zero. Explain the need for the parentheses around the numerator.

16.3 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.

16.4 In Section 6.5.1 (page 257) 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.)

16.5 Beginning with the translation of the previous exercise, reapply the code improvements discussed in this chapter to the `combinations` subroutine.

16.6 Give an example in which the numbered heuristics listed on page ◎354 do not lead to an optimal code schedule.

16.7 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).

16.8 In the sidebar on page 774, we noted two additional properties (other than definite assignment) that a Java Virtual Machine must verify in order to protect itself from potentially erroneous byte code. 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.

16.9 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.

16.10 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 vi depends on the value of register vj or of variable x, then during available expression analysis we say that vi $\in Kill_B$ if B contains an assignment to vj or x and does not contain a subsequent assignment to vi.)

16.11 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.

16.12 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.

16.13 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.

16.14 Should live variable analysis be performed before or after loop invariant elimination (or should it be done twice, before *and* after)? Justify your answer.

16.15 Starting with the naive `gcd` code of Figure 1.6 (page 34), show the result of local redundancy elimination (via value numbering) and instruction scheduling.

16.16 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?

16.17 In Section 16.6 (page 352) we noted that hardware register renaming can often hide anti- and output dependences. Will it help in Figure 16.12? Explain.

16.18 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.

16.19 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.

16.20 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?

16.21 Using the code in Figures ◎16.11 and ◎16.13 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 ◎16.15)?

16.22 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 ◎16.14, 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?

16.23 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.

16.24 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.

16.25 Modify the tiled matrix transpose of Example ◎16.32 to eliminate the min calculations in the bounds of the inner loops. Perform the same modification on your answer to Exercise ◎16.23.

16.11 Explorations

16.26 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?

16.27 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?

16.28 A recent trend is the use of static compiler analysis 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 15.17) 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 15.18)? Can static analysis be useful if it has both false negatives (errors it misses) and false positives (correct code it flags as erroneous)?

16.29 In a somewhat gloomy parody of Moore's Law, Todd Proebsting of Microsoft Research (himself an eminent compiler researcher) offers *Proebsting's Law*: "Compiler advances double computing power every 18 years." (See *research.microsoft.com/ toddpro/* for pointers.)

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?

16.12 Bibliographic Notes

Recent compiler textbooks (e.g., those of Cooper and Torczon [CT04], Grune et al. [GBJL01], or Appel [App97]) 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; KKR+86; Pey87; Pey92; WM95, Chap. 3; App97, Chap. 15; GBJL01, Chap. 7]; object-oriented [AH95; GDDC97; WM95, Chap. 5; App97, Chap. 14; GBJL01, Chap. 6]; and logic languages [DRSS96; FSS83; Zho96; WM95, Chap. 4; GBJL01, 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 are the most difficult part of this analysis, but significant progress has been made in recent years; Hind [Hin01] presents a comprehensive and accessible survey. Instruction scheduling from basic-block dependence DAGs is described by Gibbons and Muchnick [GM86]. The general technique is known as *list scheduling*; modern treatments appear in the texts of Muchnick [Muc97, Sec. 17.1.2] and Cooper and Torczon [CT04, 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].

Sources of information on loop transformations and parallelization include the recent text of Allen and Kennedy [AK02], the older 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].