Chapter 5

Intermediate Representations

CHAPTER OVERVIEW

The central data structure in a compiler is the intermediate form of the program being compiled. Most passes in the compiler read and manipulate the IR form of the code. Thus, decisions about what to represent and how to represent it play a crucial role in both the cost of compilation and its effectiveness. This chapter presents a survey of IR forms that compilers use, including graphical IR, linear IRS, and symbol tables.

Keywords: Intermediate Representation, Graphical IR, Linear IR, SSA Form, Symbol Table

5.1 INTRODUCTION

Compilers are typically organized as a series of passes. As the compiler derives knowledge about the code it compiles, it must convey that information from one pass to another. Thus, the compiler needs a representation for all of the facts that it derives about the program. We call this representation an *intermediate representation*, or IR. A compiler may have a single IR, or it may have a series of IRS that it uses as it transforms the code from source language into its target language. During translation, the IR form of the input program is the definitive form of the program. The compiler does not refer back to the source text; instead, it looks to the IR form of the code. The properties of a compiler's IR or IRS have a direct effect on what the compiler can do to the code.

Almost every phase of the compiler manipulates the program in its IR form. Thus, the properties of the IR, such as the mechanisms for reading and writing specific fields, for finding specific facts or annotations, and for navigating around a program in IR form, have a direct impact on the ease of writing the individual passes and on the cost of executing those passes.

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00005-0

Conceptual Roadmap

This chapter focuses on the issues that surround the design and use of an IR in compilation. Section 5.1.1 provides a taxonomic overview of IRs and their properties. Many compiler writers consider trees and graphs as the natural representation for programs; for example, parse trees easily capture the derivations built by a parser. Section 5.2 describes several IRs based on trees and graphs. Of course, most processors that compilers target have linear assembly languages as their native language. Accordingly, some compilers use linear IRs with the rationale that those IRs expose properties of the target machine's code that the compiler should explicitly see. Section 5.3 examines linear IRs.

Appendix B.4 provides more material on symbol table implementation.

The final sections of this chapter deal with issues that relate to IRs but are not, strictly speaking, IR design issues. Section 5.4 explores issues that relate to naming: the choice of specific names for specific values. Naming can have a strong impact on the kind of code generated by a compiler. That discussion includes a detailed look at a specific, widely used IR called *static single-assignment form*, or SSA. Section 5.5 provides a high-level overview of how the compiler builds, uses, and maintains *symbol tables*. Most compilers build one or more symbol tables to hold information about names and values and to provide efficient access to that information.

Overview

To convey information between its passes, a compiler needs a representation for all of the knowledge that it derives about the program being compiled. Thus, almost all compilers use some form of intermediate representation to model the code being analyzed, translated, and optimized. Most passes in the compiler consume IR; the scanner is an exception. Most passes in the compiler produce IR; passes in the code generator can be exceptions. Many modern compilers use multiple IRs during the course of a single compilation. In a pass-structured compiler, the IR serves as the primary and definitive representation of the code.

A compiler's IR must be expressive enough to record all of the useful facts that the compiler might need to transmit between passes. Source code is insufficient for this purpose; the compiler derives many facts that have no representation in source code, such as the addresses of variables and constants or the register in which a given parameter is passed. To record all of the detail that the compiler must encode, most compiler writers augment the IR with tables and sets that record additional information. We consider these tables part of the IR.

Selecting an appropriate IR for a compiler project requires an understanding of the source language, the target machine, and the properties of the applications that the compiler will translate. For example, a source-to-source translator might use an IR that closely resembles the source code, while a compiler that produces assembly code for a microcontroller might obtain better results with an assembly-code-like IR. Similarly, a compiler for C might need annotations about pointer values that are irrelevant in a compiler for Perl, and a Java compiler keeps records about the class hierarchy that have no counterpart in a C compiler.

Implementing an IR forces the compiler writer to focus on practical issues. The compiler needs inexpensive ways to perform the operations that it does frequently. It needs concise ways to express the full range of constructs that might arise during compilation. The compiler writer also needs mechanisms that let humans examine the IR program easily and directly. Self-interest should ensure that compiler writers pay heed to this last point. Finally, compilers that use an IR almost always make multiple passes over the IR for a program. The ability to gather information in one pass and use it in another improves the quality of code that a compiler can generate.

The ⇒ symbol in ILOC serves no purpose except to improve readability.

5.1.1 A Taxonomy of Intermediate Representations

Compilers have used many kinds of IR. We will organize our discussion of IRs along three axes: structural organization, level of abstraction, and naming discipline. In general, these three attributes are independent; most combinations of organization, abstraction, and naming have been used in some compiler.

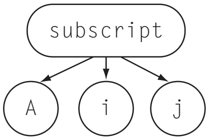
Broadly speaking, IRs fall into three structural categories:

- *Graphical IRs* encode the compiler's knowledge in a graph. The algorithms are expressed in terms of graphical objects: nodes, edges, lists, or trees. The parse trees used to depict derivations in Chapter 3 are a graphical IR.
- *Linear IRs* resemble pseudo-code for some abstract machine. The algorithms iterate over simple, linear sequences of operations. The ILOC code used in this book is a form of linear IR.
- *Hybrid IRs* combine elements of both graphical and linear IRs, in an attempt to capture their strengths and avoid their weaknesses. A common hybrid representation uses a low-level linear IR to represent blocks of straight-line code and a graph to represent the flow of control among those blocks.

The structural organization of an IR has a strong impact on how the compiler writer thinks about analysis, optimization, and code generation. For example, treelike IRs lead naturally to passes structured as some form of treewalk. Similarly, linear IRs lead naturally to passes that iterate over the operations in order.

The second axis of our IR taxonomy is the level of abstraction at which the IR represents operations. The IR can range from a near-source representation in which a single node might represent an array access or a procedure call to a low-level representation in which several IR operations must be combined to form a single target-machine operation.

To illustrate the possibilities, assume that `A[1...10,1...10]` is an array of four-byte elements stored in row-major order and consider how the compiler might represent the array reference `A[i,j]` in a source-level tree and in ILOC.



Source-Level Tree

```
subI   r_i, 1    ⇒ r_1
multI  r_1, 10   ⇒ r_2
subI   r_j, 1    ⇒ r_3
add    r_2, r_3  ⇒ r_4
multI  r_4, 4    ⇒ r_5
loadI  @A        ⇒ r_6
add    r_5, r_6  ⇒ r_7
load   r_7       ⇒ r_Aij
```

ILOC Code

In the source-level tree, the compiler can easily recognize the computation as an array reference; the ILOC code obscures that fact fairly well. In a compiler that tries to determine when two different references can touch the same memory location, the source-level tree makes it easy to find and compare references. By contrast, the ILOC code makes those tasks hard. Optimization only makes the situation worse; in the ILOC code, optimization might move parts of the address computation elsewhere. The tree node will remain intact under optimization.

On the other hand, if the goal is to optimize the target-machine code generated for the array access, the ILOC code lets the compiler optimize details that remain implicit in the source-level tree. For this purpose, a low-level IR may prove better.

Not all tree-based IRs use a near-source-level of abstraction. To be sure, parse trees are implicitly related to the source code, but trees with other levels of abstraction have been used in many compilers. Many C compilers, for example, have used low-level expression trees. Similarly, linear IRs can have relatively high-level constructs, such as a `max` or a `min` operator, or a string-copy operation.

The third axis of our IR taxonomy deals with the name space used to represent values in the code. In translating source code to a lower-level form, the compiler must choose names for a variety of distinct values. For example, to evaluate `a - 2×b` in a low-level IR, the compiler might generate a sequence of operations such as those shown in the margin. Here, the compiler has used four names, $\mathtt{t}_1$ through $\mathtt{t}_4$. An equally valid scheme would replace the occurrences of $\mathtt{t}_2$ and $\mathtt{t}_4$ with $\mathtt{t}_1$, which cuts the number of names in half.

$$\begin{aligned}
\mathtt{t}_1 &\leftarrow \mathtt{b} \\
\mathtt{t}_2 &\leftarrow 2 \times \mathtt{t}_1 \\
\mathtt{t}_3 &\leftarrow \mathtt{a} \\
\mathtt{t}_4 &\leftarrow \mathtt{t}_3 - \mathtt{t}_2
\end{aligned}$$

The choice of a naming scheme has a strong effect on how optimization can improve the code. If the subexpression `2 - b` has a unique name, the compiler might find other evaluations of `2 - b` that it can replace with a reference to the value produced here. If the name is reused, the current value may not be available at the subsequent, redundant evaluation. The choice of a naming scheme also has an impact on compile time, because it determines the sizes of many compile-time data structures.

As a practical matter, the costs of generating and manipulating an IR should concern the compiler writer, since they directly affect a compiler's speed. The data-space requirements of different IRs vary over a wide range. Since the compiler typically touches all of the space that it allocates, data space usually has a direct relationship to running time. To make this discussion concrete, consider the IRs used in two different research systems that we built at Rice University.

- The $\mathcal{R}^n$ Programming Environment built an abstract syntax tree for FORTRAN. Nodes in the tree occupied 92 bytes each. The parser built an average of eleven nodes per FORTRAN source line, for a size of just over 1,000 bytes per source-code line.
- The MSCP research compiler used a full-scale implementation of ILOC. (The ILOC in this book is a simple subset.) ILOC operations occupy 23 to 25 bytes. The compiler generates an average of roughly fifteen ILOC operations per source-code line, or about 375 bytes per source-code line. Optimization reduces the size to just over three operations per source-code line, or fewer than 100 bytes per source-code line.

Finally, the compiler writer should consider the expressiveness of the IR—its ability to accommodate all the facts that the compiler needs to record. The IR for a procedure might include the code that defines it, the results of static analysis, profile data from previous executions, and maps to let the debugger understand the code and its data. All of these facts should be expressed in a way that makes clear their relationship to specific points in the IR.

5.2 GRAPHICAL IRS

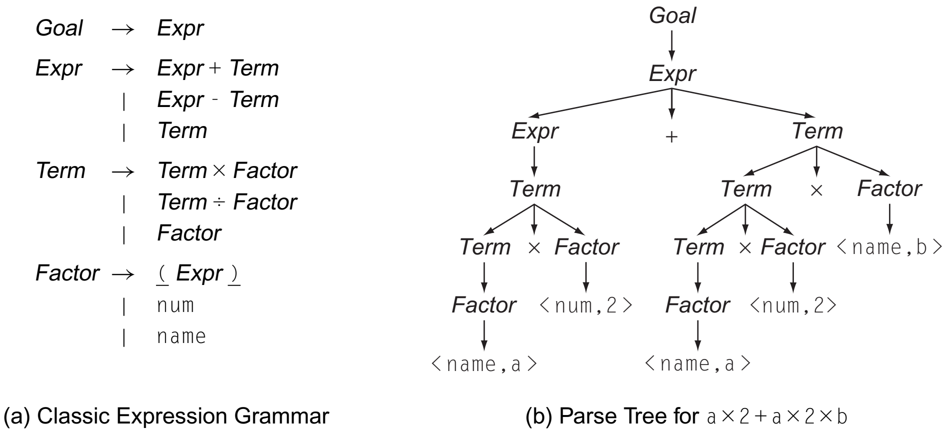
Many compilers use IRs that represent the underlying code as a graph. While all the graphical IRs consist of nodes and edges, they differ in their level of abstraction, in the relationship between the graph and the underlying code, and in the structure of the graph.

5.2.1 Syntax-Related Trees

The parse trees shown in Chapter 3 are graphs that represent the source-code form of the program. Parse trees are one specific form of treelike IRs. In most treelike IRs, the structure of the tree corresponds to the syntax of the source code.

Parse Trees

As we saw in Section 3.2.2, the *parse tree* is a graphical representation for the derivation, or parse, that corresponds to the input program. Figure 5.1 shows the classic expression grammar alongside a parse tree for a × 2 + a × 2 × b. The parse tree is large relative to the source text because it represents the complete derivation, with a node for each grammar symbol in the derivation. Since the compiler must allocate memory for each node and each edge, and it must traverse all those nodes and edges during compilation, it is worth considering ways to shrink this parse tree.



(a) Classic Expression Grammar

(b) Parse Tree for a × 2 + a × 2 × b

■ **FIGURE 5.1** Parse Tree for a × 2 + a × 2 × b Using the Classic Expression Grammar.

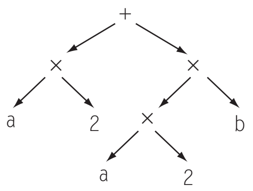
Minor transformations on the grammar, as described in Section 3.6.1, can eliminate some of the steps in the derivation and their corresponding syntax-tree nodes. A more effective technique is to abstract away those nodes that serve no real purpose in the rest of the compiler. This approach leads to a simplified version of the parse tree, called an abstract syntax tree.

Parse trees are used primarily in discussions of parsing, and in attribute-grammar systems, where they are the primary IR. In most other applications in which a source-level tree is needed, compiler writers tend to use one of the more concise alternatives, described in the remainder of this subsection.

Abstract Syntax Trees

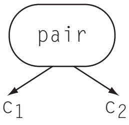
Abstract syntax tree
An AST is a contraction of the parse tree that omits most nodes for nonterminal symbols.

The *abstract syntax tree* (AST) retains the essential structure of the parse tree but eliminates the extraneous nodes. The precedence and meaning of the expression remain, but extraneous nodes have disappeared. Here is the AST for `a × 2 + a × 2 × b`:

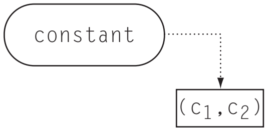


The AST is a near-source-level representation. Because of its rough correspondence to a parse tree, the parser can built an AST directly (see Section 4.4.2).

ASTs have been used in many practical compiler systems. Source-to-source systems, including syntax-directed editors and automatic parallelization tools, often use an AST from which source code can easily be regenerated. The S-expressions found in Lisp and Scheme implementations are, essentially, ASTs.



AST Designed for Editing



AST for Compiling

Even when the AST is used as a near-source-level representation, representation choices affect usability. For example, the AST in the $\mathcal{R}^n$ Programming Environment used the subtree shown in the margin to represent a `complex` constant in FORTRAN, written (c_1,c_2). This choice worked well for the syntax-directed editor, in which the programmer was able to change c_1 and c_2 independently; the `pair` node corresponded to the parentheses and the comma.

This pair format, however, proved problematic for the compiler. Each part of the compiler that dealt with constants needed special-case code for complex constants. All other constants were represented with a single

STORAGE EFFICIENCY AND GRAPHICAL REPRESENTATIONS

Many practical systems have used abstract syntax trees to represent the source text being translated. A common problem encountered in these systems is the size of the AST relative to the input text. Large data structures can limit the size of programs that the tools can handle.

The AST nodes in the $\mathcal{R}^n$ Programming Environment were large enough that they posed a problem for the limited memory systems of 1980s workstations. The cost of disk I/O for the trees slowed down all the $\mathcal{R}^n$ tools.

No single problem leads to this explosion in AST size. $\mathcal{R}^n$ had only one kind of node, so that structure included all the fields needed by any node. This simplified allocation but increased the node size. (Roughly half the nodes were leaves, which need no pointers to children.) In other systems, the nodes grow through the addition of myriad minor fields used by one pass or another in the compiler. Sometimes, the node size increases over time, as new features and passes are added.

Careful attention to the form and content of the AST can shrink its size. In $\mathcal{R}^n$, we built programs to analyze the contents of the AST and how the AST was used. We combined some fields and eliminated others. (In some cases, it was less expensive to recompute information than to write it and read it.) In a few cases, we used hash linking to record unusual facts—using one bit in the field that stores each node's type to indicate the presence of additional information stored in a hash table. (This scheme reduced the space devoted to fields that were rarely used.) To record the AST on disk, we converted it to a linear representation with a preorder treewalk; this eliminated the need to record any internal pointers.

In $\mathcal{R}^n$, these changes reduced the size of ASTs in memory by roughly 75 percent. On disk, after the pointers were removed, the files were about half the size of their memory representation. These changes let $\mathcal{R}^n$ handle larger programs and made the tools more responsive.

node that contained a pointer to the constant's actual text. Using a similar format for complex constants would have complicated some operations, such as editing the complex constants and loading them into registers. It would have simplified others, such as comparing two constants. Taken over the entire system, the simplifications would likely have outweighed the complications.

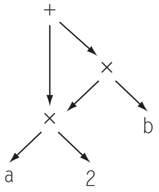
Abstract syntax trees have found widespread use. Many compilers and interpreters use them; the level of abstraction that those systems need varies widely. If the compiler generates source code as its output, the AST typically has source-level abstractions. If the compiler generates assembly code, the final version of the AST is usually at or below the abstraction level of the machine's instruction set.

Directed Acyclic Graphs

Directed acyclic graph
A DAG is an AST with sharing. Identical subtrees are instantiated once, with multiple parents.

While the AST is more concise than a syntax tree, it faithfully retains the structure of the original source code. For example, the AST for a × 2 + a × 2 × b contains two distinct copies of the expression a × 2. A *directed acyclic graph* (DAG) is a contraction of the AST that avoids this duplication. In a DAG, nodes can have multiple parents, and identical subtrees are reused. Such sharing makes the DAG more compact than the corresponding AST.

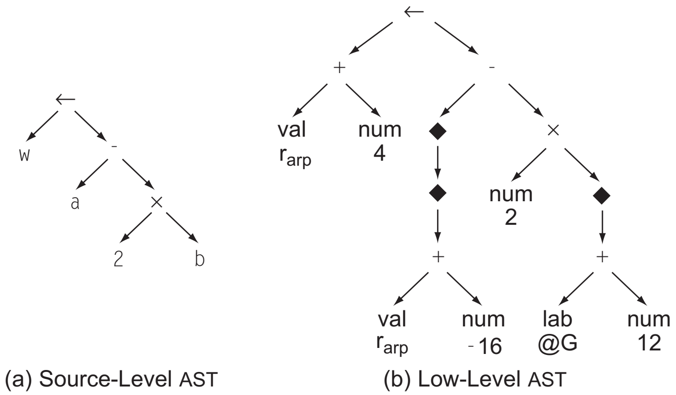
For expressions without assignment, textually identical expressions must produce identical values. The DAG for a × 2 + a × 2 × b, shown to the left, reflects this fact by sharing a single copy of a × 2. The DAG encodes an explicit hint for evaluating the expression. If the value of a cannot change between the two uses of a, then the compiler should generate code to evaluate a × 2 once and use the result twice. This strategy can reduce the cost of evaluation. However, the compiler must prove that a's value cannot change. If the expression contains neither assignment nor calls to other procedures, the proof is easy. Since an assignment or a procedure call can change the value associated with a name, the DAG construction algorithm must invalidate subtrees as the values of their operands change.



DAGs are used in real systems for two reasons. If memory constraints limit the size of programs that the compiler can handle, using a DAG can help by reducing the memory footprint. Other systems use DAGs to expose redundancies. Here, the benefit lies in better compiled code. These latter systems tend to use the DAG as a derivative IR—building the DAG, transforming the definitive IR to reflect the redundancies, and discarding the DAG.

Level of Abstraction

All of our example trees so far have shown near-source IRS. Compilers also use low-level trees. Tree-based techniques for optimization and code generation, in fact, may require such detail. As an example, consider the statement w ← a - 2 × b. A source-level AST creates a concise form, as shown in Figure 5.2a. However, the source-level tree lacks much of the detail needed to translate the statement into assembly code. A low-level tree, shown in Figure 5.2b, can make that detail explicit. This tree introduces four new node types. A val node represents a value already in a register. A num node represents a known constant. A lab node represents an assembly-level label, typically a relocatable symbol. Finally, ◆ is an operator that dereferences a value; it treats the value as a memory address and returns the contents of the memory at that address.



■ **FIGURE 5.2** Abstract Syntax Trees with Different Levels of Abstraction.

Data area
The compiler groups together storage for values that have the same lifetime and visibility. We call these blocks of storage *data areas*.

The low-level tree reveals the address calculations for the three variables. w is stored at offset 4 from the pointer in r_{arp}, which holds the pointer to the data area for the current procedure. The double dereference of a shows that it is a call-by-reference formal parameter accessed through a pointer stored 16 bytes before r_{arp}. Finally, b is stored at offset 12 after the label @G.

The level of abstraction matters because the compiler can, in general, only optimize details that are exposed in the IR. Properties that are implicit in the IR are hard to change, in part because the compiler would need to translate implicit facts in different, instance-specific ways. For example, to customize the code generated for an array reference, the compiler must rewrite the related IR expressions. In a real program, different array references are optimized in different ways, each according to the surrounding context. For the compiler to tailor those references, it must be able to write down the improvements in the IR.

As a final point, notice that the representations for the variable references in the low-level tree reflect the different interpretations that occur on the right and left side of the assignment. On the left-hand side, w evaluates to an address, while both a and b evaluate to values because of the ◆ operator.

5.2.2 Graphs

While trees provide a natural representation for the grammatical structure of the source code discovered by parsing, their rigid structure makes them less useful for representing other properties of programs. To model these aspects of program behavior, compilers often use more general graphs as IRs. The DAG introduced in the previous section is one example of a graph.

Control-Flow Graph

The simplest unit of control flow in a program is a *basic block*—a maximal length sequence of straightline, or branch-free, code. A basic block is a sequence of operations that always execute together, unless an operation raises an exception. Control always enters a basic block at its first operation and exits at its last operation.

Basic block
a maximal-length sequence of branch-free code

It begins with a labelled operation and ends with a branch, jump, or predicated operation.

A *control-flow graph* (CFG) models the flow of control between the basic blocks in a program. A CFG is a directed graph, $G = (N, E)$. Each node $n \in N$ corresponds to a basic block. Each edge $e = (n_i, n_j) \in E$ corresponds to a possible transfer of control from block n_i to block n_j.

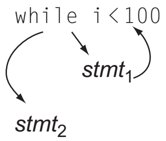
Control-flow graph
A CFG has a node for every basic block and an edge for each possible control transfer between blocks.

We use the acronym CFG for both *context-free grammar* (see page 86) and *control-flow graph*. The meaning should be clear from context.

To simplify the discussion of program analysis in Chapters 8 and 9, we assume that each CFG has a unique entry node, n_0, and a unique exit node, n_f. In the CFG for a procedure, n_0 corresponds to the procedure's entry point. If a procedure has multiple entries, the compiler can insert a unique n_0 and add edges from n_0 to each actual entry point. Similarly, n_f corresponds to the procedure's exit. Multiple exits are more common than multiple entries, but the compiler can easily add a unique n_f and connect each of the actual exits to it.

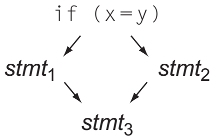
The CFG provides a graphical representation of the possible runtime control-flow paths. The CFG differs from the syntax-oriented IRS, such as an AST, in which the edges show grammatical structure. Consider the following CFG for a while loop:

```
while(i<100)
   begin
      stmt1
      end
stmt2
```



The edge from $stmt_1$ back to the loop header creates a cycle; the AST for this fragment would be acyclic. For an if-then-else construct, the CFG is acyclic:

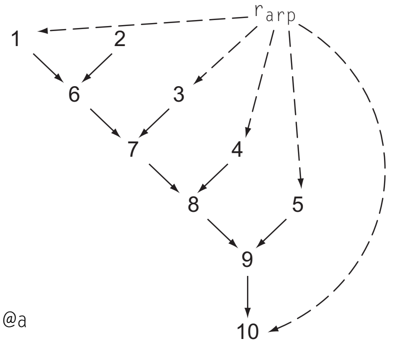
```
if (x=y)
   then stmt1
   else stmt2
stmt3
```



It shows that control always flows from $stmt_1$ and $stmt_2$ to $stmt_3$. In an AST, that connection is implicit, rather than explicit.

Compilers typically use a CFG in conjunction with another IR. The CFG represents the relationships among blocks, while the operations inside a block

```
1   loadAI  rarp,@a ⇒ ra
2   loadI   2       ⇒ r2
3   loadAI  rarp,@b ⇒ rb
4   loadAI  rarp,@c ⇒ rc
5   loadAI  rarp,@d ⇒ rd
6   mult    ra,r2   ⇒ ra
7   mult    ra,rb   ⇒ ra
8   mult    ra,rc   ⇒ ra
9   mult    ra,rd   ⇒ ra
10  storeAI ra      ⇒ rarp,@a
```



■ **FIGURE 5.3** An ILOC Basic Block and Its Dependence Graph.

are represented with another IR, such as an expression-level AST, a DAG, or one of the linear IRs. The resulting combination is a hybrid IR.

Single-statement blocks
a block of code that corresponds to a single source-level statement

Some authors recommend building CFGs in which each node represents a shorter segment of code than a basic block. The most common alternative block is a *single-statement block*. Using single-statement blocks can simplify algorithms for analysis and optimization.

The tradeoff between a CFG built with single-statement blocks and one built with basic blocks revolves around time and space. A CFG built on single-statement blocks has more nodes and edges than a CFG built with basic blocks. The single-statement version uses more memory and takes longer to traverse than the basic-block version of a CFG. More important, as the compiler annotates the nodes and edges in the CFG, the single-statement CFG has many more sets than the basic-block CFG. The time and space spent in constructing and using these annotations undoubtedly dwarfs the cost of CFG construction.

Many parts of the compiler rely on a CFG, either explicitly or implicitly. Analysis to support optimization generally begins with control-flow analysis and CFG construction (Chapter 9). Instruction scheduling needs a CFG to understand how the scheduled code for individual blocks flows together (Chapter 12). Global register allocation relies on a CFG to understand how often each operation might execute and where to insert loads and stores for spilled values (Chapter 13).

Dependence Graph

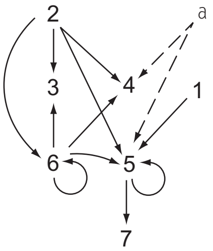
Data-dependence graph
a graph that models the flow of values from definitions to uses in a code fragment

Compilers also use graphs to encode the flow of values from the point where a value is created, a *definition*, to any point where it is used, a *use*. A *data-dependence graph* embodies this relationship. Nodes in a data-dependence

```
1   x ← 0
2   i ← 1

3   while (i < 100)
4     if (a[i] > 0)
5        then x ← x + a[i]
6     i ← i + 1

7   print x
```



■ **FIGURE 5.4** Interaction between Control Flow and the Dependence Graph.

graph represent operations. Most operations contain both definitions and uses. An edge in a data-dependence graph connects two nodes, one that defines a value and another that uses it. We draw dependence graphs with edges that run from definition to use.

To make this concrete, Figure 5.3 reproduces the example from Figure 1.3 and shows its data-dependence graph. The graph has a node for each statement in the block. Each edge shows the flow of a single value. For example, the edge from 3 to 7 reflects the definition of r_b in statement 3 and its subsequent use in statement 7. r_{arp} contains the starting address of the local data area. Uses of r_{arp} refer to its implicit definition at the start of the procedure; they are shown with dashed lines.

The edges in the graph represent real constraints on the sequencing of operations—a value cannot be used until it has been defined. However, the dependence graph does not fully capture the program's control flow. For example, the graph requires that 1 and 2 precede 6. Nothing, however, requires that 1 or 2 precedes 3. Many execution sequences preserve the dependences shown in the code, including $\langle 1,2,3,4,5,6,7,8,9,10\rangle$ and $\langle 2,1,6,3,7,4,8,5,9,10\rangle$. The freedom in this partial order is precisely what an "out-of-order" processor exploits.

At a higher level, consider the code fragment shown in Figure 5.4. References to a[i] are shown deriving their values from a node representing prior definitions of a. This connects all uses of a together through a single node. Without sophisticated analysis of the subscript expressions, the compiler cannot differentiate between references to individual array elements.

This dependence graph is more complex than the previous example. Nodes 5 and 6 both depend on themselves; they use values that they may have defined in a previous iteration. Node 6, for example, can take the value of i from either 2 (in the initial iteration) or from itself (in any subsequent iteration). Nodes 4 and 5 also have two distinct sources for the value of i: nodes 2 and 6.

Data-dependence graphs are often used as a derivative IR—constructed from the definitive IR for a specific task, used, and then discarded. They play a central role in instruction scheduling (Chapter 12). They find application in a variety of optimizations, particularly transformations that reorder loops to expose parallelism and to improve memory behavior; these typically require sophisticated analysis of array subscripts to determine more precisely the patterns of access to arrays. In more sophisticated applications of the data-dependence graph, the compiler may perform extensive analysis of array subscript values to determine when references to the same array can overlap.

Call Graph

Interprocedural
Any technique that examines interactions across multiple procedures is called *interprocedural*.

Intraprocedural
Any technique that limits its attention to a single procedure is called *intraprocedural*.

Call graph
a graph that represents the calling relationships among the procedures in a program

The call graph has a node for each procedure and an edge for each call site.

To address inefficiencies that arise across procedure boundaries, some compilers perform *interprocedural* analysis and optimization. To represent the runtime transfers of control between procedures, compilers use a *call graph*. A call graph has a node for each procedure and an edge for each distinct procedure call site. Thus, the code calls q from three textually distinct sites in p; the call graph has three edges (p, q), one for each call site.

Both software-engineering practice and language features complicate the construction of a call graph.

- Separate compilation, the practice of compiling small subsets of a program independently, limits the compiler's ability to build a call graph and to perform interprocedural analysis and optimization. Some compilers build partial call graphs for all of the procedures in a compilation unit and perform analysis and optimization across that set. To analyze and optimize the whole program in such a system, the programmer must present it all to the compiler at once.
- Procedure-valued parameters, both as input parameters and as return values, complicate call-graph construction by introducing ambiguous call sites. If `fee` takes a procedure-valued argument and invokes it, that site has the potential to call a different procedure on each invocation of `fee`. The compiler must perform an interprocedural analysis to limit the set of edges that such a call induces in the call graph.
- Object-oriented programs with inheritance routinely create ambiguous procedure calls that can only be resolved with additional type information. In some languages, interprocedural analysis of the class hierarchy can provide the information needed to disambiguate these calls. In other languages, that information cannot be known until runtime. Runtime resolution of ambiguous calls poses a serious problem for call graph construction; it also creates significant runtime overheads on the execution of the ambiguous calls.

Section 9.4 discusses practical techniques for call graph construction.

SECTION REVIEW

Graphical IRs present an abstract view of the code being compiled. They differ in the meaning imputed to each node and each edge.

- In a parse tree, nodes represent syntactic elements in the source-language grammar, while the edges tie those elements together into a derivation.
- In an abstract syntax tree or a dag, nodes represent concrete items from the source-language program, and edges tie those together in a way that indicates control-flow relationships and the flow of data.
- In a control-flow graph, nodes represent blocks of code and edges represent transfers of control between blocks. The definition of a block may vary, from a single statement through a basic block.
- In a dependence graph, the nodes represent computations and the edges represent the flow of values from definitions to uses; as such, edges also imply a partial order on the computations.
- In a call graph, the nodes represent individual procedures and the edges represent individual call sites. Each call site has a distinct edge to provide a representation for call-site specific knowledge, such as parameter bindings.

Graphical IRs encode relationships that may be difficult to represent in a linear IR. A graphical IR can provide the compiler with an efficient way to move between logically connected points in the program, such as the definition of a variable and its use, or the source of a conditional branch and its target.

Review Questions

1. Compare and contrast the difficulty of writing a *prettyprinter* for a parse tree, an AST and a DAG. What additional information would be needed to reproduce the original code's format precisely?
2. How does the number of edges in a dependence graph grow as a function of the input program's size?

Prettyprinter
a program that walks a syntax tree and writes out the original code

5.3 LINEAR IRS

The alternative to a graphical IR is a linear IR. An assembly-language program is a form of linear code. It consists of a sequence of instructions that execute in their order of appearance (or in an order consistent with that order). Instructions may contain more than one operation; if so, those operations execute in parallel. The linear IRs used in compilers resemble the assembly code for an abstract machine.

The logic behind using a linear form is simple. The source code that serves as input to the compiler is a linear form, as is the target-machine code that it emits. Several early compilers used linear IRs; this was a natural notation for their authors, since they had previously programmed in assembly code.

Linear IRs impose a clear and useful ordering on the sequence of operations. For example, in Figure 5.3, contrast the ILOC code with the data-dependence graph. The ILOC code has an implicit order; the dependence graph imposes a partial ordering that allows many different execution orders.

If a linear IR is used as the definitive representation in a compiler, it must include a mechanism to encode transfers of control among points in the program. Control flow in a linear IR usually models the implementation of control flow on the target machine. Thus, linear codes usually include conditional branches and jumps. Control flow demarcates the basic blocks in a linear IR; blocks end at branches, at jumps, or just before labelled operations.

Taken branch
In most ISAs, conditional branches use one label. Control flows either to the label, called the *taken branch*, or to the operation that follows the label, called the *not-taken* or *fall-through branch*.

In the ILOC used throughout this book, we include a branch or jump at the end of every block. In ILOC, the branch operations specify a label for both the taken path and the not-taken path. This eliminates any fall-through paths at the end of a block. Together, these stipulations make it easier to find basic blocks and to reorder them.

Many kinds of linear IRs have been used in compilers.

Destructive operation
an operation in which one of the operands is always redefined with the result

- One-address codes model the behavior of accumulator machines and stack machines. These codes expose the machine's use of implicit names so that the compiler can tailor the code for it. The resulting code is quite compact.
- Two-address codes model a machine that has destructive operations. These codes fell into disuse as memory constraints became less important; a three-address code can model destructive operations explicitly.
- Three-address codes model a machine where most operations take two operands and produce a result. The rise of RISC architectures in the 1980s and 1990s made these codes popular, since they resemble a simple RISC machine.

The remainder of this section describes two linear IRs that remain popular: stack-machine code and three-address code. Stack-machine code offers a compact, storage-efficient representation. In applications where IR size matters, such as a Java applet transmitted over a network before execution, stack-machine code makes sense. Three-address code models the instruction format of a modern RISC machine; it has distinct names for two operands and a result. You are already familiar with one three-address code: the ILOC used in this book.

5.3.1 Stack-Machine Code

Stack-machine code, a form of one-address code, assumes the presence of a stack of operands. Most operations take their operands from the stack and push their results back onto the stack. For example, an integer subtract operation would remove the top two elements from the stack and push their difference onto the stack. The stack discipline creates a need for some new operations. Stack IRS usually include a `swap` operation that interchanges the top two elements of the stack. Several stack-based computers have been built; this IR seems to have appeared in response to the demands of compiling for these machines. Stack-machine code for the expression `a - 2×b` appears in the margin.

```
push  2
push  b
multiply
push  a
subtract
```

Stack-Machine Code

Stack-machine code is compact. The stack creates an implicit name space and eliminates many names from the IR. This shrinks the size of a program in IR form. Using the stack, however, means that all results and arguments are transitory, unless the code explicitly moves them to memory.

Stack-machine code is simple to generate and to execute. Smalltalk 80 and Java both use bytecodes, a compact IR similar in concept to stack-machine code. The bytecodes either run in an interpreter or are translated into target-machine code just prior to execution. This creates a system with a compact form of the program for distribution and a reasonably simple scheme for porting the language to a new target machine (implementing the interpreter).

Bytecode
an IR designed specifically for its compact form; typically code for an abstract stack machine

The name derives from its limited size; opcodes are limited to one byte or less.

5.3.2 Three-Address Code

In three-address code most operations have the form `i ← j op k`, with an operator (`op`), two operands (`j` and `k`) and one result (`i`). Some operators, such as an immediate load and a jump, will need fewer arguments. Sometimes, an operation with more than three addresses is needed. Three address code for `a - 2×b` appears in the margin. ILOC is another example of a three-address code.

$$
\begin{aligned}
t_1 &\leftarrow 2\\
t_2 &\leftarrow b\\
t_3 &\leftarrow t_1 \times t_2\\
t_4 &\leftarrow a\\
t_5 &\leftarrow t_4 - t_3
\end{aligned}
$$

Three-Address Code

Three-address code is attractive for several reasons. First, three-address code is reasonably compact. Most operations consist of four items: an operation and three names. Both the operation and the names are drawn from limited sets. Operations typically require 1 or 2 bytes. Names are typically represented by integers or table indices; in either case, 4 bytes is usually enough. Second, separate names for the operands and the target give the compiler freedom to control the reuse of names and values; three-address code has no destructive operations. Three-address code introduces a new set of compiler-generated names—names that hold the results of the various operations. A carefully chosen name space can reveal new opportunities to improve the code. Finally, since many modern processors implement three-address operations, a three-address code models their properties well.

Within three-address codes, the set of specific supported operators and their level of abstraction can vary widely. Often, a three-address IR will contain mostly low-level operations, such as jumps, branches, and simple memory operations, alongside more complex operations that encapsulate control flow, such as `max` or `min`. Representing these complex operations directly makes them easier to analyze and optimize.

For example, `mvcl` (move characters long) takes a source address, a destination address, and a character count. It copies the specified number of characters from memory beginning at the source address to memory beginning at the destination address. Some machines, like the IBM 370, implement this functionality in a single instruction (`mvcl` is a 370 opcode). On machines that do not implement the operation in hardware, it may require many operations to perform such a copy.

Adding `mvcl` to the three-address code lets the compiler use a compact representation for this complex operation. It allows the compiler to analyze, optimize, and move the operation without concern for its internal workings. If the hardware supports an `mvcl`-like operation, then code generation will map the IR construct directly to the hardware operation. If the hardware does not, then the compiler can translate `mvcl` into a sequence of lower-level IR operations or a procedure call before final optimization and code generation.

5.3.3 Representing Linear Codes

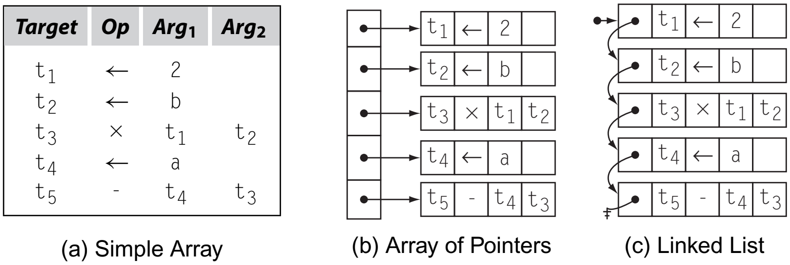
Many data structures have been used to implement linear IRs. The choices that a compiler writer makes affect the costs of various operations on IR code. Since a compiler spends most of its time manipulating the IR form of the code, these costs deserve some attention. While this discussion focuses on three-address codes, most of the points apply equally to stack-machine code (or any other linear form).

Three-address codes are often implemented as a set of quadruples. Each quadruple is represented with four fields: an operator, two operands (or sources), and a destination. To form blocks, the compiler needs a mechanism to connect individual quadruples. Compilers implement quadruples in a variety of ways.

$$\begin{array}{lcl} t_1 & \leftarrow & 2 \\ t_2 & \leftarrow & b \\ t_3 & \leftarrow & t_1 \times t_2 \\ t_4 & \leftarrow & a \\ t_5 & \leftarrow & t_4 - t_3 \end{array}$$

Three-Address Code

Figure 5.5 shows three different schemes for implementing the three-address code for `a - 2×b`, repeated in the margin. The simplest scheme, in



Target	Op	Arg_1	Arg_2
t_1	←	2	
t_2	←	b	
t_3	×	t_1	t_2
t_4	←	a	
t_5	-	t_4	t_3

(a) Simple Array

(b) Array of Pointers

(c) Linked List

■ **FIGURE 5.5** Implementations of Three-Address Code for a - 2 × b.

Figure 5.5a, uses a short array to represent each basic block. Often, the compiler writer places the array inside a node in the CFG. (This may be the most common form of hybrid IR.) The scheme in Figure 5.5b uses an array of pointers to group quadruples into a block; the pointer array can be contained in a CFG node. The final scheme, in Figure 5.5c, links the quadruples together to form a list. It requires less storage in the CFG node, at the cost of restricting accesses to sequential traversals.

Consider the costs incurred in rearranging the code in this block. The first operation loads a constant into a register; on most machines this translates directly into an immediate load operation. The second and fourth operations load values from memory, which on most machines might incur a multicycle delay unless the values are already in the primary cache. To hide some of the delay, the instruction scheduler might move the loads of b and a in front of the immediate load of 2.

In the simple array scheme, moving the load of b ahead of the immediate load requires saving the four fields of the first operation, copying the corresponding fields from the second slot into the first slot, and overwriting the fields in the second slot with the saved values for the immediate load. The array of pointers requires the same three-step approach, except that only the pointer values must be changed. Thus, the compiler saves the pointer to the immediate load, copies the pointer to the load of b into the first slot in the array, and overwrites the second slot in the array with the saved pointer to the immediate load. For the linked list, the operations are similar, except that the complier must save enough state to let it traverse the list.

Now, consider what happens in the front end when it generates the initial round of IR. With the simple array form and the array of pointers, the compiler must select a size for the array—in effect, the number of quadruples that it expects in a block. As it generates the quadruples, it fills in the array. If the array is too large, it wastes space. If it is too small, the compiler must

INTERMEDIATE REPRESENTATIONS IN ACTUAL USE

In practice, compilers use a variety of IRs. Legendary FORTRAN compilers of yore, such as IBM's FORTRAN H compilers, used a combination of quadruples and control-flow graphs to represent the code for optimization. Since FORTRAN H was written in FORTRAN, it held the IR in an array.

For a long time, GCC relied on a very low-level IR, called register transfer language (RTL). In recent years, GCC has moved to a series of IRs. The parsers initially produce a near-source tree; these trees can be language specific but are required to implement parts of a common interface. That interface includes a facility for lowering the trees to the second IR, GIMPLE. Conceptually, GIMPLE consists of a language-independent, tree-like structure for control-flow constructs, annotated with three-address code for expressions and assignments. It is designed, in part, to simplify analysis. Much of GCC's new optimizer uses GIMPLE; for example, GCC builds static single-assignment form on top of GIMPLE. Ultimately, GCC translates GIMPLE into RTL for final optimization and code generation.

The LLVM compiler uses a single low-level IR; in fact, the name LLVM stands for "low-level virtual machine." LLVM's IR is a linear three-address code. The IR is fully typed and has explicit support for array and structure addresses. It provides support for vector or SIMD data and operations. Scalar values are maintained in SSA form throughout the compiler. The LLVM environment uses GCC front ends, so LLVM IR is produced by a pass that performs GIMPLE-to-LLVM translation.

The Open64 compiler, an open-source compiler for the IA-64 architecture, uses a family of five related IRs, called WHIRL. The initial translation in the parser produces a near-source-level WHIRL. Subsequent phases of the compiler introduce more detail to the WHIRL program, lowering the level of abstraction toward the actual machine code. This lets the compiler use a source-level AST for dependence-based transformations on the source text and a low-level IR for the late stages of optimization and code generation.

reallocate it to obtain a larger array, copy the contents of the "too small" array into the new, larger array, and deallocate the small array. The linked list, however, avoids these problems. Expanding the list just requires allocating a new quadruple and setting the appropriate pointer in the list.

A multipass compiler may use different implementations to represent the IR at different points in the compilation process. In the front end, where the focus is on generating the IR, a linked list might both simplify the implementation and reduce the overall cost. In an instruction scheduler, with its focus on rearranging the operations, either of the array implementations might make more sense.

Notice that some information is missing from Figure 5.5. For example, no labels are shown because labels are a property of the block rather than any individual quadruple. Storing a list of labels with the block saves space in each quadruple; it also makes explicit the property that labels occur only on the first operation in a basic block. With labels attached to a block, the compiler can ignore them when reordering operations inside the block, avoiding one more complication.

5.3.4 Building a Control-Flow Graph from a Linear Code

Compilers often must convert between different IRS, often different styles of IRS. One routine conversion is to build a CFG from a linear IR such as ILOC. The essential features of a CFG are that it identifies the beginning and end of each basic block and connects the resulting blocks with edges that describe the possible transfers of control among blocks. Often, the compiler must build a CFG from a simple, linear IR that represents a procedure.

As a first step, the compiler must find the beginning and the end of each basic block in the linear IR. We will call the initial operation of a block a *leader*. An operation is a leader if it is the first operation in the procedure, or if it has a label that is, potentially, the target of some branch. The compiler can identify leaders in a single pass over the IR, shown in Figure 5.6a. It iterates over the operations in the program, in order, finds the labelled statements, and records them as leaders.

If the linear IR contains labels that are not used as branch targets, then treating labels as leaders may unnecessarily split blocks. The algorithm could

Ambiguous jump
a branch or jump whose target cannot be determined at compile time; typically, a jump to an address in a register

```
next ← 1
Leader[next++] ← 1
for i ← 1 to n
    if $op_i$ has a label $l_i$ then
        Leader[next++] ← i
        create a CFG node for $l_i$
```

(a) Finding Leaders

```
for i ← 1 to next - 1
    j ← Leader[i] + 1
    while (j ≤ n and $op_j \notin$ Leader)
        j ← j + 1
    j ← j - 1
    Last[i] ← j
    if $op_j$ is "cbr $r_k \rightarrow l_1, l_2$" then
        add edge from j to node for $l_1$
        add edge from j to node for $l_2$
    else if $op_j$ is "jumpI $\rightarrow l_1$" then
        add edge from j to node for $l_1$
    else if $op_j$ is "jump $\rightarrow r_1$" then
        add edges from j to all labelled statements
```

(b) Finding Last and Adding Edges

■ **FIGURE 5.6** Building a Control-Flow Graph.

COMPLICATIONS IN CFG CONSTRUCTION

Features of the IR, the target machine, and the source language can complicate CFG construction.

Ambiguous jumps may force the compiler to introduce edges that are never feasible at runtime. The compiler writer can improve this situation by including features in the IR that record potential jump targets. ILOC includes the `tbl` pseudo-operation to let the compiler record the potential targets of an ambiguous jump. Anytime the compiler generates a `jump`, it should follow the `jump` with a set of `tbl` operations that record the possible branch targets. CFG construction can use these hints to avoid spurious edges.

If the compiler builds a CFG from target-machine code, features of the target architecture can complicate the process. The algorithm in Figure 5.6 assumes that all leaders, except the first, are labelled. If the target machine has fall-through branches, the algorithm must be extended to recognize unlabeled statements that receive control on a fall-through path. PC-relative branches cause a similar set of problems.

Branch delay slots introduce several problems. A labelled statement that sits in a branch delay slot is a member of two distinct blocks. The compiler can cure this problem by replication—creating new (unlabeled) copies of the operations in the delay slots. Delay slots also complicate finding the end of a block. The compiler must place operations located in delay slots into the block that precedes the branch or jump.

If a branch or jump can occur in a branch delay slot, the CFG builder must walk forward from the leader to find the block-ending branch—the first branch it encounters. Branches in the delay slot of a block-ending branch can, themselves, be pending on entry to the target block. They can split the target block and force creation of new blocks and new edges. This kind of behavior seriously complicates CFG construction.

Some languages allow jumps to labels outside the current procedure. In the procedure containing the branch, the branch target can be modelled with a new CFG node created for that purpose. The complication arises on the other end of the branch. The compiler must know that the target label is the target of a nonlocal branch, or else subsequent analysis may produce misleading results. For this reason, languages such as Pascal or Algol restricted nonlocal gotos to labels in visible outer lexical scopes. C requires the use of the functions `setjmp` and `longjmp` to expose these transfers.

track which labels are jump targets. However, if the code contains any ambiguous jumps, then it must treat all labelled statements as leaders anyway.

The second pass, shown in Figure 5.6b, finds every block-ending operation. It assumes that every block ends with a branch or a jump and that branches specify labels for both outcomes—a "branch taken" label and a "branch not taken" label. This simplifies the handling of blocks and allows the compiler's back end to choose which path will be the "fall through" case of a branch. (For the moment, assume branches have no delay slots.)

To find the end of each block, the algorithm iterates through the blocks, in order of their appearance in the `Leader` array. It walks forward through the IR until it finds the leader of the next block. The operation immediately before that leader ends the current block. The algorithm records that operation's index in `Last[i]`, so that the pair ⟨`Leader[i]`, `Last[i]`⟩ describes block *i*. It adds edges to the CFG as needed.

For a variety of reasons, the CFG should have a unique entry node n_0 and a unique exit node n_f. The underlying code should have this shape. If it does not, a simple postpass over the graph can create n_0 and n_f.

SECTION REVIEW

Linear IRs represent the code being compiled as an ordered sequence of operations. Linear IRs can vary in their level of abstraction; the source code for a program in a plain text file is a linear form, as is the assembly code for that same program. Linear IRs lend themselves to compact, human-readable representations.

Two widely used linear IRs are bytecodes, generally implemented as a one-address code with implicit names on many operations, and three-address code, generally implemented as a set of binary operations that have distinct name fields for two operands and one result.

Review Questions

1. Consider the expression $a \times 2 + a \times 2 \times b$. Translate it into stack machine code and into three address code. Compare and contrast the number of operations and the number of operands in each form. How do they compare against the trees in Figure 5.1?
2. Sketch an algorithm to build control-flow graphs from ILOC for programs that include spurious labels and ambiguous jumps.

5.4 MAPPING VALUES TO NAMES

The choice of a specific IR and a level of abstraction helps determine what operations the compiler can manipulate and optimize. For example, a source-level AST makes it easy to find all the references to an array ×. At the same time, it hides the details of the address calculations required to access an element of ×. In contrast, a low-level, linear IR such as ILOC exposes the details of the address calculation, at the cost of obscuring the fact that a specific reference relates to ×.

Similarly, the discipline that the compiler uses to assign internal names to the various values computed during execution has an effect on the code that it can generate. A naming scheme can expose opportunities for optimization or it can obscure them. The compiler must invent names for many, if not all, of the intermediate results that the program produces when it executes. The choices that it makes with regard to names determines, to a large extent, which computations can be analyzed and optimized.

5.4.1 Naming Temporary Values

The IR form of a program usually contains more detail than does the source version. Some of those details are implicit in the source code; others result from deliberate choices in the translation. To see this, consider the four-line block of source code shown in Figure 5.7a. Assume that the names refer to distinct values.

The block deals with just four names, { a, b, c, d }. It refers to more than four values. Each of b, c, and d have a value before the first statement executes. The first statement computes a new value, b + c, as does the second, which computes a - d. The expression b + c in the third statement computes

```
a ← b + c
b ← a - d
c ← b + c
d ← a - d
```

(a) Source Code

```
t1 ← b
t2 ← c
t3 ← t1 + t2
a  ← t3
t4 ← d
t1 ← t3 - t4
b  ← t1
t2 ← t1 + t2
c  ← t2
t4 ← t3 - t4
d  ← t4
```

(b) Source Names

```
t1 ← b
t2 ← c
t3 ← t1 + t2
a  ← t3
t4 ← d
t5 ← t3 - t4
b  ← t5
t6 ← t5 + t2
c  ← t6
t5 ← t3 - t4
d  ← t5
```

(c) Value Names

■ **FIGURE 5.7** Naming Leads to Different Translations.

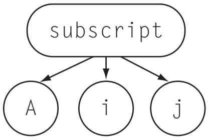
a different value than the earlier b+c, unless c=d initially. Finally, the last statement computes a - d; its result is always identical to that produced by the second statement.

The source code names tell the compiler little about the values that they hold. For example, the use of b in the first and third statements refer to distinct values (unless c=d). The reuse of the name b conveys no information; in fact, it might mislead a casual reader into thinking that the code sets a and c to the same value.

When the compiler names each of these expressions, it can chose names in ways that specifically encode useful information about their values. Consider, for example, the translations shown in Figures 5.7b and 5.7c. These two variants were generated with different naming disciplines.

The code in Figure 5.7b uses fewer names than the code in 5.7c. It follows the source code names, so that a reader can easily relate the code back to the code in Figure 5.7a. The code in Figure 5.7c uses more names than the code in 5.7b. Its naming discipline reflects the computed values and ensures that textually identical expressions produce the same result. This scheme makes it obvious that a and c may receive different values, while b and d must receive the same value.

As another example of the impact of names, consider again the representation of an array reference, A[i,j]. Figure 5.8 shows two IR fragments that represent the same computation at very different levels of abstraction. The high-level IR, in Figure 5.8a, contains all the essential information and is easy to identify as a subscript reference. The low-level IR, in Figure 5.8b,



```
load   1      ⇒ r1
sub    rj, r1 ⇒ r2
loadI  10     ⇒ r3
mult   r2, r3 ⇒ r4
sub    ri, r1 ⇒ r5
add    r4, r5 ⇒ r6
loadI  @A     ⇒ r7
add    r7, r6 ⇒ r8
load   r8     ⇒ rAij
```

(a) Source-Level Abstract Syntax Tree

(b) Low-Level Linear Code (ILOC)

■ **FIGURE 5.8** Different Levels of Abstraction for an Array Subscript Reference .

exposes many details to the compiler that are implicit in the high-level AST fragment. All of the details in the low-level IR can be inferred from the source-level AST.

In the low-level IR, each intermediate result has its own name. Using distinct names exposes those results to analysis and transformation. In practice, most of the improvement that compilers achieve in optimization arises from capitalizing on context. To make that improvement possible, the IR must expose the context. Naming can hide context, as when it reuses one name for many distinct values. It can also expose context, as when it creates a correspondence between names and values. This issue is not specifically a property of linear codes; the compiler could use a lower-level AST that exposed the entire address computation.

5.4.2 Static Single-Assignment Form

SSA form
an IR that has a value-based name system, created by renaming and use of pseudo-operations called ϕ-functions

SSA encodes both control and value flow. It is used widely in optimization (see Section 9.3).

Static single-assignment form (SSA) is a naming discipline that many modern compilers use to encode information about both the flow of control and the flow of data values in the program. In SSA form, names correspond uniquely to specific definition points in the code; each name is defined by one operation, hence the name static single assignment. As a corollary, each use of a name as an argument in some operation encodes information about where the value originated; the textual name refers to a specific definition point. To reconcile this single-assignment naming discipline with the effects of control flow, SSA form inserts special operations, called ϕ-functions, at points where control-flow paths meet.

ϕ-function
A ϕ-function takes several names and merges them, defining a new name.

A program is in SSA form when it meets two constraints: (1) each definition has a distinct name; and (2) each use refers to a single definition. To transform an IR program to SSA form, the compiler inserts ϕ-functions at points where different control-flow paths merge and it then renames variables to make the single-assignment property hold.

To clarify the impact of these rules, consider the small loop shown on the left side of Figure 5.9. The right column shows the same code in SSA form. Variable names include subscripts to create a distinct name for each definition. ϕ-functions have been inserted at points where multiple distinct values can reach the start of a block. Finally, the `while` construct has been rewritten with two distinct tests, to reflect the fact that the initial test refers to x_0 while the end-of-loop test refers to x_2.

The ϕ-function's behavior depends on context. It defines its target SSA name with the value of its argument that corresponds to the edge along which

```
x ← ···
y ← ···
while(x < 100)
  x ← x + 1
  y ← y + x
```

(a) Original Code

```
      x0 ← ···
      y0 ← ···
      if (x0 ≥ 100) goto next
loop: x1 ← φ(x0,x2)
      y1 ← φ(y0,y2)
      x2 ← x1 + 1
      y2 ← y1 + x2
      if (x2 < 100) goto loop
next: x3 ← φ(x0,x2)
      y3 ← φ(y0,y2)
```

(b) Code in SSA Form

■ **FIGURE 5.9** A Small Loop in SSA Form.

control entered the block. Thus, when control flows into the loop from the block above the loop, the ϕ-functions at the top of the loop body copy the values of x_0 and y_0 into x_1 and y_1, respectively. When control flows into the loop from the test at the loop's bottom, the ϕ-functions select their other arguments, x_2 and y_2.

On entry to a basic block, all of its ϕ-functions execute concurrently, before any other statement. First, they all read the values of the appropriate arguments, then they all define their target SSA names. Defining their behavior in this way allows the algorithms that manipulate SSA form to ignore the ordering of ϕ-functions at the top of a block—an important simplification. It can complicate the process of translating SSA form back into executable code, as we shall see in Section 9.3.5.

SSA form was intended for code optimization. The placement of ϕ-functions in SSA form encodes information about both the creation of values and their uses. The single-assignment property of the name space allows the compiler to sidestep many issues related to the lifetimes of values; for example, because names are never redefined or killed, the value of a name is available along any path that proceeds from that operation. These two properties simplify and improve many optimization techniques.

The example exposes some oddities of SSA form that bear explanation. Consider the ϕ-function that defines x_1. Its first argument, x_0, is defined in the block that precedes the loop. Its second argument, x_2, is defined later in the block labelled `loop`. Thus, when the ϕ first executes, one of its arguments is undefined. In many programming-language contexts, this would cause problems. Since the ϕ-function reads only one argument, and that argument

THE IMPACT OF NAMING

In the late 1980s, we experimented with naming schemes in a FORTRAN compiler. The first version generated a new temporary register for each computation by bumping a simple counter. It produced large name spaces, for example, 985 names for a 210-line implementation of the singular value decomposition (SVD). The name space seemed large for the program size. It caused speed and space problems in the register allocator, where the size of the name space governs the size of many data structures. (Today, we have better data structures and faster machines with more memory.)

The second version used an allocate/free protocol to manage names. The front end allocated temporaries on demand and freed them when the immediate uses were finished. This scheme used fewer names; for example, SVD used roughly 60 names. It sped up allocation, reducing, for example, the time to find live variables in SVD by 60 percent.

Unfortunately, associating multiple expressions with a single temporary name obscured the flow of data and degraded the quality of optimization. The decline in code quality overshadowed any compile-time benefits.

Further experimentation led to a short set of rules that yielded strong optimization while mitigating growth in the name space.

1. Each textual expression received a unique name, determined by entering the operator and operands into a hash table. Thus, each occurrence of an expression, for example, $r_{17}+r_{21}$, targeted the same register.
2. In ⟨op⟩ $r_i, r_j \Rightarrow r_k$, k was chosen so that $i, j < k$.
3. Register copy operations (i2i $r_i \Rightarrow r_j$ in ILOC) were allowed to have $i > j$ only if r_j corresponded to a scalar program variable. The registers for such variables were only defined by copy operations. Expressions evaluated into their "natural" register and then were moved into the register for the variable.
4. Each store operation (store $r_i \Rightarrow r_j$ in ILOC) is followed by a copy from r_i into the variable's named register. (Rule 1 ensures that loads from that location always target the same register. Rule 4 ensures that the virtual register and memory location contain the same value.)

This name-space scheme used about 90 names for SVD, but exposed all the optimizations found with the first name-space scheme. The compiler used these rules until we adopted SSA form, with its discipline for names.

corresponds to the most recently taken edge in the CFG, it can never read the undefined value.

ϕ-functions do not conform to a three-address model. A ϕ-function takes an arbitrary number of operands. To fit SSA form into a three-address IR, the

BUILDING SSA

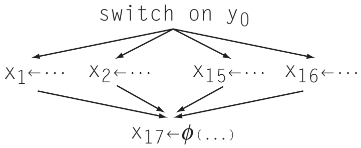
Static single-assignment form is the only IR we describe that does not have an obvious construction algorithm. Section 9.3 presents the algorithm in detail. However, a sketch of the construction process will clarify some of the issues. Assume that the input program is already in ILOC form. To convert it to an equivalent linear form of SSA, the compiler must first insert ϕ-functions and then rename the ILOC virtual registers.

The simplest way to insert ϕ-functions adds one for each ILOC virtual register at the start of each basic block that has more than one predecessor in the control-flow graph. This inserts many unneeded ϕ-functions; most of the complexity in the full algorithm is aimed at reducing the number of extraneous ϕ-functions.

To rename the ILOC virtual registers, the compiler can process the blocks, in a depth-first order. For each virtual register, it keeps a counter. When the compiler encounters a definition of r_i, it increments the counter for r_i, say to k, and rewrites the definition with the name r_{i_k}. As the compiler traverses the block, it rewrites each use of r_i with r_{i_k} until it encounters another definition of r_i. (That definition bumps the counter to $k+1$.) At the end of a block, the compiler looks down each control-flow edge and rewrites the appropriate ϕ-function parameter for r_i in each block that has multiple predecessors.

After renaming, the code conforms to the two rules of SSA form. Each definition creates a unique name. Each use refers to a single definition. Several better SSA construction algorithms exist; they insert fewer ϕ-functions than this simple approach.

compiler writer must include a mechanism for representing operations with longer operand lists. Consider the block at the end of a case statement as shown in the margin.


switch on y_0
$x_1 \leftarrow \cdots$
$x_2 \leftarrow \cdots$
$x_{15} \leftarrow \cdots$
$x_{16} \leftarrow \cdots$
$x_{17} \leftarrow \phi(\ldots)$

The ϕ-function for x_{17} must have an argument for each case. A ϕ-operation has one argument for each entering control-flow path; thus, it does not fit into the fixed-arity, three-address scheme.

In a simple array representation for three-address code, the compiler writer must either use multiple slots for each ϕ-operation or use a side data structure to hold the ϕ-operations' arguments. In the other two schemes for implementing three-address code shown in Figure 5.5, the compiler can insert tuples of varying size. For example, the tuples for load and load immediate might have space for just two names, while the tuple for a ϕ-operation could be large enough to accommodate all its operands.

5.4.3 Memory Models

Just as the mechanism for naming temporary values affects the information that can be represented in an IR version of a program, so, too, does the compiler's choice of a storage location for each value. The compiler must determine, for each value computed in the code, where that value will reside. For the code to execute, the compiler must assign a specific location, such as register r_{13} or 16 bytes from the label L0089. Before the final stages of code generation, however, the compiler may use symbolic addresses that encode a level in the memory hierarchy, for example, registers or memory, but not a specific location within that level.

Consider the ILOC examples used throughout this book. A symbolic memory address is denoted by prefixing it with the character @. Thus, @x is the offset of × from the start of the storage area containing it. Since r_{arp} holds the activation record pointer, an operation that uses @x and r_{arp} to compute an address depends, implicitly, on the decision to store the variable x in the memory reserved for the current procedure's activation record.

In general, compilers work from one of two memory models.

1. *Register-to-Register Model* Under this model, the compiler keeps values in registers aggressively, ignoring any limitations imposed by the size of the machine's physical register set. Any value that can legally be kept in a register for most of its lifetime is kept in a register. Values are stored to memory only when the semantics of the program require it—for example, at a procedure call, any local variable whose address is passed as a parameter to the called procedure must be stored back to memory. A value that cannot be kept in a register for most of its lifetime is stored in memory. The compiler generates code to store its value each time it is computed and to load its value at each use.
2. *Memory-to-Memory Model* Under this model, the compiler assumes that all values are kept in memory locations. Values move from memory to a register just before they are used. Values move from a register to memory just after they are defined. The number of registers named in the IR version of the code can be small compared to the register-to-register model. In this model, the designer may find it worthwhile to include memory-to-memory operations, such as a memory-to-memory add, in the IR.

The choice of memory model is mostly orthogonal to the choice of IR. The compiler writer can build a memory-to-memory AST or a memory-to-memory version of ILOC just as easily as register-to-register versions of either of these

THE HIERARCHY OF MEMORY OPERATIONS IN ILOC 9X

The ILOC used in this book is abstracted from an IR named ILOC 9X that was used in a research compiler project at Rice University. ILOC 9X includes a hierarchy of memory operations that the compiler uses to encode knowledge about values. At the bottom of the hierarchy, the compiler has little or no knowledge about the value; at the top of the hierarchy, it knows the actual value. These operations are as follows:

Operation	Meaning
Immediate load	Loads a known constant value into a register.
Nonvarying load	Loads a value that does not change during execution. The compiler does not know the value, but can prove that it is not defined by a program operation.
Scalar load & store	Operate on a scalar value, not an array element, a structure element, or a pointer-based value.
General load & store	Operate on a value that may be an array element, a structure element, or a pointer-based value. This is the general-case operation.

By using this hierarchy, the front end can encode knowledge about the target value directly into the ILOC 9X code. As other passes discover additional information, they can rewrite operations to change a value from using a general-purpose load to a more restricted form. If the compiler discovers that some value is a known constant, it can replace a general load or a scalar load of that value with an immediate load. If an analysis of definitions and uses discovers that some location cannot be defined by any executable store operation, loads of that value can be rewritten to use a non-varying load.

Optimizations can capitalize on the knowledge encoded in this fashion. For example, a comparison between the result of a non-varying load and a constant must itself be invariant—a fact that might be difficult or impossible to prove with a scalar load or a general load.

IRS. (Stack-machine code and code for an accumulator machine might be exceptions; they contain their own unique memory models.)

The choice of memory model has an impact on the rest of the compiler. With a register-to-register model, the compiler typically uses more registers than the target machine provides. Thus, the register allocator must map the set of *virtual registers* used in the IR program onto the physical registers provided by the target machine. This often requires insertion of extra load, store, and copy operations, making the code slower and larger. With a memory-to-memory model, however, the IR version of the code typically uses fewer registers than a modern processor provides. Here, the register allocator looks for memory-based values that it can hold in registers for longer periods of time. In this model, the allocator makes the code faster and smaller by removing loads and stores.

Compilers for RISC machines tend to use the register-to-register model for two reasons. First, the register-to-register model more closely reflects the instruction sets of RISC architectures. RISC machines do not have a full complement of memory-to-memory operations; instead, they implicitly assume that values can be kept in registers. Second, the register-to-register model allows the compiler to encode directly in the IR some of the subtle facts that it derives. The fact that a value is kept in a register means that the compiler, at some earlier point, had proof that keeping it in a register is safe. Unless it encodes that fact in the IR, the compiler will need to prove it, again and again.

To elaborate, if the compiler can prove that only one name provides access to a value, it can keep that value in a register. If multiple names might exist, the compiler must behave conservatively and keep the value in memory. For example, a local variable x can be kept in a register, unless it can be referenced in another scope. In a language that supports nested scopes, like Pascal or Ada, this reference can occur in a nested procedure. In C, this can occur if the program takes x's address, &x, and accesses the value through that address. In Algol or PL/I, the program can pass x as a call-by-reference parameter to another procedure.

SECTION REVIEW

The schemes used to name values in a compiler's IR have a direct effect on the compiler's ability to optimize the IR and to generate quality assembly code from the IR. The compiler must generate internal names for all values, from variables in the source language program to the intermediate values computed as part of an address expression for a subscripted array reference. Careful use of names can encode and expose facts for late use in optimization; at the same time, proliferation of names can slow the compiler by forcing it to use larger data structures.

The name space generated in SSA form has gained popularity because it encodes useful properties; for example, each name corresponds to a unique definition in the code. This precision can aid in optimization, as we will see in Chapter 8.

The name space can also encode a memory model. A mismatch between the memory model and the target machine's instruction set can complicate subsequent optimization and code generation, while a close match allows the compiler to tailor carefully to the target machine.

Review Questions

1. Consider the function `fib` shown in the margin. Write down the ILOC that a compiler's front end might generate for this code under a register-to-register model and under a memory-to-memory model. How do the two compare? Under what circumstances might each memory be desirable?
2. Convert the register-to-register code that you generated in the previous question into SSA form. Are there ϕ-functions whose output value can never be used?

```
int fib(int n) {
  int x = 1;
  int y = 1;
  int z = 1;
  while(n > 1)
     z = x + y;
     x = y;
     y = z;
     n = n - 1;
  return z;
}
```

5.5 SYMBOL TABLES

As part of translation, a compiler derives information about the various entities manipulated by the program being translated. It must discover and store many distinct kinds of information. It encounters a wide variety of names—variables, defined constants, procedures, functions, labels, structures, and files. As discussed in the previous section, the compiler also generates many names. For a variable, it needs a data type, its storage class, the name and lexical level of its declaring procedure, and a base address and offset in memory. For an array, the compiler also needs the number of dimensions and the upper and lower bounds for each dimension. For records or structures, it needs a list of the fields, along with the relevant information for each field. For functions and procedures, it needs the number of parameters and their types, as well as the types of any returned values; a more sophisticated translation might record information about what variables a procedure can reference or modify.

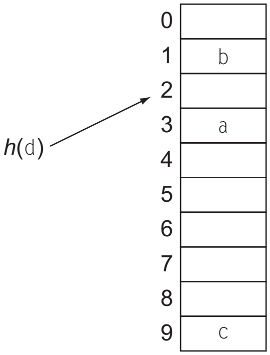
The compiler must either record this information in the IR or re-derive it on demand. For the sake of efficiency, most compilers record facts rather than recompute them. These facts can be recorded directly in the IR. For example, a compiler that builds an AST might record information about variables as annotations (or attributes) of the node representing each variable's declaration. The advantage of this approach is that it uses a single representation for the code being compiled. It provides a uniform access method and a single implementation. The disadvantage of this approach is that the single access method may be inefficient—navigating the AST to find the appropriate declaration has its own costs. To eliminate this inefficiency, the compiler can thread the IR so that each reference has a link back to the corresponding declaration. This adds space to the IR and overhead to the IR builder.

When the compiler writes the IR to disk, it may be cheaper to recompute facts than to write them and then read them.

The alternative, as we saw in Chapter 4, is to create a central repository for these facts and provide efficient access to it. This central repository, called a symbol table, becomes an integral part of the compiler's IR. The symbol table localizes information derived from potentially distant parts of the source code. It makes such information easily and efficiently available, and it simplifies the design and implementation of any code that must refer to information about variables derived earlier in compilation. It avoids the expense of searching the IR to find the portion that represents a variable's declaration; using a symbol table often eliminates the need to represent the declarations directly in the IR. (An exception occurs in source-to-source translation. The compiler may build a symbol table for efficiency and preserve the declaration syntax in the IR so that it can produce an output program that closely resembles the input program.) It eliminates the overhead of making each reference contain a pointer to the declaration. It replaces both of these with a computed mapping from the textual name to the stored information. Thus, in some sense, the symbol table is simply an efficiency trick.

At many places in this text, we refer to "the symbol table." As we shall see in Section 5.5.4, the compiler may include several distinct, specialized symbol tables. A careful implementation might use the same access methods for all these tables.

Symbol-table implementation requires attention to detail. Because nearly every aspect of translation refers to the symbol table, efficiency of access is critical. Because the compiler cannot predict, before translation, the number of names that it will encounter, expanding the symbol table must be both graceful and efficient. This section provides a high-level treatment of the issues that arise in designing a symbol table. It presents the compiler-specific aspects of symbol-table design and use. For deeper implementation details and design alternatives, see Section B.4 in Appendix B.



5.5.1 Hash Tables

A compiler accesses its symbol table frequently. Thus, efficiency is a key issue in the design of a symbol table. Because hash tables provide constant-time expected-case lookups, they are the method of choice for implementing symbol tables. Hash tables are conceptually elegant. They use a *hash function*, h, to map names to small integers, and use the small integer to index the table. With a hashed symbol table, the compiler stores all the information that it derives about the name n in the table in slot $h(n)$. The figure in the margin shows a simple ten-slot hash table. It is a vector of records, each record holding the compiler-generated description of a single name. The names a, b, and c have already been inserted. The name d is being inserted, at $h(\mathtt{d}) = 2$.

The primary reason to use hash tables is to provide a constant-time expected-case lookup keyed by a textual name. To achieve this, h must be inexpensive to compute. Given an appropriate function h, accessing the record for n requires computing $h(n)$ and indexing into the table at $h(n)$. If h maps two or more symbols to the same small integer, a "collision" occurs. (In the marginal figure, this would occur if $h(\texttt{d}) = 3$.) The implementation must handle this situation gracefully, preserving both the information and the lookup time. In this section, we assume that h is a perfect hash function, that is, it never produces a collision. Furthermore, we assume that the compiler knows, in advance, how large to make the table. Appendix B.4 describes hash-table implementation in more detail, including hash functions, collision handling, and schemes for expanding a hash table.

Hash tables can be used as an efficient representation for sparse graphs. Given two nodes, x and y, an entry for the key xy indicates that an edge (x, y) exists. (This scheme requires a hash function that generates a good distribution from a pair of small integers; both the multiplicative and universal hash functions described in Appendix B.4.1 work well.) A well-implemented hash table can provide fast insertion and a fast test for the presence of a specific edge. Additional information is required to answer questions such as "What nodes are adjacent to x?"

5.5.2 **Building a Symbol Table**

The symbol table defines two interface routines for the rest of the compiler.

1. *LookUp(name)* returns the record stored in the table at h(*name*) if one exists. Otherwise, it returns a value indicating that *name* was not found.
2. *Insert(name,record)* stores the information in *record* in the table at h(*name*). It may expand the table to accommodate the record for *name*.

The compiler can use separate functions for *LookUp* and *Insert*, or they can be combined by passing *LookUp* a flag that specifies whether or not to insert the name. This ensures, for example, that a *LookUp* of an undeclared variable will fail—a property useful for detecting a violation of the declare-before-use rule in syntax-directed translation schemes or for supporting nested lexical scopes.

This simple interface fits directly into the ad hoc syntax-directed translation schemes described in Chapter 4. In processing declaration syntax, the compiler builds up a set of attributes for each variable. When the parser recognizes a production that declares some variable, it can enter the name and

AN ALTERNATIVE TO HASHING

Hashing is the method most widely used to organize a compiler's symbol table. Multiset discrimination is an interesting alternative that eliminates any possibility of worst-case behavior. The critical insight behind multiset discrimination is that the index can be constructed offline in the scanner.

To use multiset discrimination, the compiler writer must take a different approach to scanning. Instead of processing the input incrementally, the compiler scans the entire program to find the complete set of identifiers. As it discovers each identifier, it creates a tuple ⟨*name,position*⟩, where *name* is the text of the identifier and *position* is its ordinal position in the list of classified words, or *tokens*. It enters all the tuples into a large set.

The next step sorts the set lexicographically. In effect, this creates a set of subsets, one per identifier. Each of these subsets holds the tuples for all the occurrences of its identifier. Since each tuple refers to a specific token, through its *position* value, the compiler can use the sorted set to modify the token stream. The compiler makes a linear scan over the set, processing each subset. It allocates a symbol-table index for the entire subset, then rewrites the tokens to include that index. This augments the identifier tokens with their symbol-table indices. If the compiler needs a textual lookup function, the resulting table is ordered alphabetically for a binary search.

The price for using this technique is an extra pass over the token stream, along with the cost of the lexicographic sort. The advantages, from a complexity perspective, are that it avoids any possibility of hashing's worst-case behavior and that it makes the initial size of the symbol table obvious, even before parsing. This technique can be used to replace a hash table in almost any application in which an offline solution will work.

attributes into the symbol table using `Insert`. If a variable name can appear in only one declaration, the parser can call `LookUp` first to detect a repeated use of the name. When the parser encounters a variable name outside the declaration syntax, it uses `LookUp` to obtain the appropriate information from the symbol table. `LookUp` fails on any undeclared name. The compiler writer, of course, may need to add functions to initialize the table, to store it to and retrieve it from external media, and to finalize it. For a language with a single name space, this interface suffices.

5.5.3 Handling Nested Scopes

Few programming languages provide a single unified name space. Most languages allow a program to declare names at multiple levels. Each of these levels has a *scope*, or a region in the program's text where the name can be used. Each of these levels has a *lifetime*, or a period at runtime where the value is preserved.

If the source language allows scopes to be nested one inside another, then the front end needs a mechanism to translate a reference, such as x, to the proper scope and lifetime. The primary mechanism that compilers use to perform this translation is a scoped symbol table.

For the purposes of this discussion, assume that a program can create an arbitrary number of scopes nested one within another. We will defer an in-depth discussion of lexical scoping until Section 6.3.1; however, most programmers have enough experience with the concept for this discussion. Figure 5.10 shows a C program that creates five distinct scopes. We will label the scopes with numbers that indicate the nesting relationships among them. The level *0* scope is the outermost scope, while the level *3* scope is the innermost one.

The table on the right side of the figure shows the names declared in each scope. The declaration of b at level *2a* hides the level *1* declaration from any code inside the block that creates level *2a*. Inside level *2b*, a reference to b again refers to the level *1* parameter. In a similar way, the declarations

```
static int w;      /* level 0 */
int x;

void example(int a, int b) {
  int c;           /* level 1 */
  {
    int b, z;      /* level 2a */
      ...
  }
  {
    int a, x;      /* level 2b */
      ...
      {
        int c, x; /* level 3 */
        b = a + b + c + w;
      }
  }
}
```

Level	Names
0	w, x, example
1	a, b, c
2a	b, z
2b	a, x
3	c, x

■ **FIGURE 5.10** Simple Lexical Scoping Example in C.

of a and × in level *2b* hide their earlier declarations (at level *1* and level *0*, respectively).

This context creates the naming environment in which the assignment statement executes. Subscripting names to show their level, we find that the assignment refers to

$$b_1 = a_{2b} + b_1 + c_3 + w_0$$

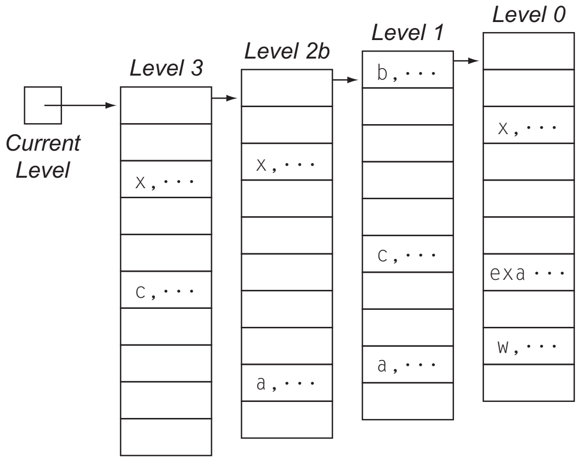
Notice that the assignment cannot use the names declared in level *2a* because that block closes, along with its scope, before level *2b* opens.

To compile a program that contains nested scopes, the compiler must map each variable reference to its specific declaration. This process, called *name resolution*, maps each reference to the lexical level at which it is declared. The mechanism that compilers use to accomplish this name resolution is a lexically scoped symbol table. The remainder of this section describes the design and implementation of lexically scoped symbol tables. The corresponding runtime mechanisms, which translate the lexical level of a reference to an address, are described in Section 6.4.3. Scoped symbol tables also have direct application in code optimization. For example, the superlocal value-numbering algorithm presented in Section 8.5.1 relies on a scoped hash table for efficiency.

The Concept

To manage nested scopes, the parser must change, slightly, its approach to symbol-table management. Each time the parser enters a new lexical scope, it can create a new symbol table for that scope. This scheme creates a sheaf of tables, linked together in an order that corresponds to the lexical nesting levels. As it encounters declarations in the scope, it enters the information into the current table. *Insert* operates on the current symbol table. When it encounters a variable reference, *LookUp* must first check the table for the current scope. If the current table does not hold a declaration for the name, it checks the table for the surrounding scope. By working its way through the symbol tables for successively lower-numbered lexical levels, it either finds the most recent declaration for the name, or fails in the outermost scope, indicating that the variable has no declaration visible in the current scope.

Figure 5.11 shows the symbol table built in this fashion for our example program, at the point where the parser has reached the assignment statement. When the compiler invokes the modified *LookUp* function for the name b, it will fail in level *3*, fail in level *2*, and find the name in level *1*. This corresponds exactly to our understanding of the program—the most recent



■ **FIGURE 5.11** Simple "Sheaf-of-Tables" Implementation.

declaration for b is as a parameter to example, at level *1*. Since the first block at level *2*, block *2a*, has already closed, its symbol table is not on the search chain. The level where the symbol is found, *1* in this case, forms the first part of an address for b. If the symbol-table record includes a storage offset for each variable, then the pair ⟨*level, offset*⟩ specifies where to find b in memory—at *offset* from the start of storage for the *level* scope. We call this pair b's *static coordinate*.

Static coordinate
a pair, <*l*,*o*>, that records address information about some variable *x*

l specifies the lexical level where *x* is declared; *o* specifies the offset within the data area for that level.

The Details

To handle this scheme, two additional calls are required. The compiler needs a call that initializes a new symbol table for a scope and one that finalizes the table for a scope.

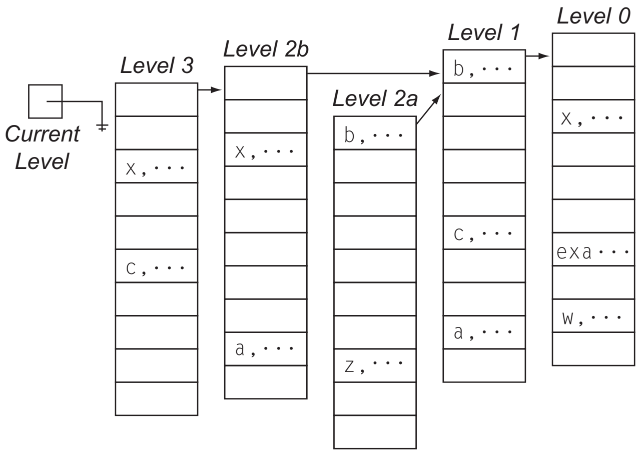
1. *InitializeScope()* increments the current level and creates a new symbol table for that level. It links the new table to the enclosing level's table and updates the current level pointer used by *LookUp* and *Insert*.
2. *FinalizeScope()* changes the current-level pointer so that it points to the table for the scope surrounding the current level and then decrements the current level. If the compiler needs to preserve the level-by-level tables for later use, *FinalizeScope* can either leave the table intact in memory or write the table to external media and reclaim its space.

To account for lexical scoping, the parser calls *InitializeScope* each time it enters a new lexical scope and *FinalizeScope* each time it exits a lexical scope. This scheme produces the following sequence of calls for the program in Figure 5.10:

1. `InitializeScope`
2. `Insert(w)`
3. `Insert(x)`
4. `Insert(example)`
5. `InitializeScope`
6. `Insert(a)`
7. `Insert(b)`
8. `Insert(c)`
9. `InitializeScope`
10. `Insert(b)`
11. `Insert(z)`
12. `FinalizeScope`
13. `InitializeScope`
14. `Insert(a)`
15. `Insert(x)`
16. `InitializeScope`
17. `Insert(c)`
18. `Insert(x)`
19. `LookUp(b)`
20. `LookUp(a)`
21. `LookUp(b)`
22. `LookUp(c)`
23. `LookUp(w)`
24. `FinalizeScope`
25. `FinalizeScope`
26. `FinalizeScope`
27. `FinalizeScope`

As it enters each scope, the compiler calls `InitializeScope`. It adds each name to the table using `Insert`. When it leaves a given scope, it calls `FinalizeScope` to discard the declarations for that scope. For the assignment statement, it looks up each of the names, as encountered. (The order of the `LookUp` calls will vary, depending on how the assignment statement is traversed.)

If `FinalizeScope` retains the symbol tables for finalized levels in memory, the net result of these calls will be the symbol table shown in Figure 5.12. The current level pointer is set to a null value. The tables for all levels are left in memory and linked together to reflect lexical nesting. The compiler can provide subsequent passes of the compiler with access to the relevant symbol-table information by storing a pointer to the appropriate table in the



■ **FIGURE 5.12** Final Table for the Example.

IR at the start of each new level. Alternatively, identifiers in the IR can point directly to their symbol-table entries.

5.5.4 The Many Uses for Symbol Tables

The preceding discussion focused on a central symbol table, albeit one that might be composed of several tables. In reality, compilers build multiple symbol tables that they use for different purposes.

Structure Table

The textual strings used to name fields in a structure or record exist in a distinct name space from the variables and procedures. The name `size` might occur in several different structures in a single program. In many programming languages, such as C or Ada, using `size` as a field in a structure does not preclude its use as a variable or function name.

For each field in a structure, the compiler needs to record its type, its size, and its offset inside the record. It gleans this information from the declarations, using the same mechanisms that it uses for processing variable declarations. It must also determine the overall size for the structure, usually computed as the sum of the field sizes, plus any overhead space required by the runtime system.

There are several approaches for managing the name space of field names:

1. *Separate Tables* The compiler can maintain a separate symbol table for each record definition. This is the cleanest idea, conceptually. If the overhead for using multiple tables is small, as in most object-oriented implementations, then using a separate table and associating it with the symbol table entry for the structure's name makes sense.
2. *Selector Table* The compiler can maintain a separate table for field names. To avoid clashes between fields with identical names in different structures, it must use qualified names—concatenate either the name of the structure or something that uniquely maps to the structure, such as the structure name's symbol-table index, to the field name. For this approach, the compiler must somehow link together the individual fields associated with each structure.
3. *Unified Table* The compiler can store field names in its principal symbol table by using qualified names. This decreases the number of tables, but it means that the principal symbol table must support all of the fields required for variables and functions, as well as all of the fields needed for each field-selector in a structure. Of the three options, this is probably the least attractive.

The separate table approach has the advantage that any scoping issues—such as reclaiming the symbol table associated with a structure—fit naturally into the scope management framework for the principal symbol table. When the structure can be seen, its internal symbol table is accessible through the corresponding structure record.

In the latter two schemes, the compiler writer will need to pay careful attention to scoping issues. For example, if the current scope declares a structure `fee` and an enclosing scope already has defined `fee`, then the scoping mechanism must correctly map `fee` to the structure (and its corresponding field entries). This may also introduce complications into the creation of qualified names. If the code contains two definitions of `fee`, each with a field named `size`, then `fee.size` is not a unique key for either field entry. This problem can be solved by associating a unique integer, generated from a global counter, with each structure name.

Linked Tables for Name Resolution in an Object-Oriented Language

In an object-oriented language, the name scoping rules are governed by the structure of the data as much as by the structure of the code. This creates a more complicated set of rules; it also leads to a more complicated set of symbol tables. Java, for example, needs tables for the code being compiled, for any external classes that are both known and referenced in the code, and for the inheritance hierarchy above the class containing the code.

A simple implementation attaches a symbol table to each class, with two nesting hierarchies: one for lexical scoping inside individual methods and the other following the inheritance hierarchy for each class. Since a single class can serve as superclass to several subclasses, this latter hierarchy is more complicated than the simple sheaf-of-tables drawing suggests. However, it is easily managed.

To resolve a name `fee` when compiling a method `m` in class C, the compiler first consults the lexically scoped symbol table for `m`. If it does not find `fee` in this table, it then searches the scopes for the various classes in the inheritance hierarchy, starting with C and proceeding up the chain of superclasses from C. If this lookup fails to find `fee`, the search then checks the global symbol table for a class or symbol table of that name. The global table must contain information on both the current package and any packages that have been used.

Thus, the compiler needs a lexically scoped table for each method, built while it compiles the methods. It needs a symbol table for each class, with links upward through the inheritance hierarchy. It needs links to the other classes in its package and to a symbol table for package-level variables. It needs access to the symbol tables for each used class. The lookup process is more complex, because it must follow these links in the correct order and examine only names that are visible. However, the basic mechanisms required to implement and manipulate the tables are already familiar.

5.5.5 Other Uses for Symbol Table Technology

The basic ideas that underlie symbol table implementation have widespread application, both inside a compiler and in other domains. Hash tables are used to implement sparse data structures; for example, a sparse array can be implemented by constructing a hash key from the indices and only storing non-zero values. Runtime systems for LISP-like languages have reduced their storage requirements by having the `cons` operator hash its arguments—effectively enforcing a rule that textually identical objects share a single instance in memory. Pure functions, those that always return the same values on the same input parameters, can use a hash table to produce an implementation that behaves as a *memo function*.

Memo function
a function that stores results in a hash table under a key built from its arguments and uses the hash table to avoid recomputation of prior results

SECTION REVIEW

Several tasks inside a compiler require efficient mappings from noninteger data into a compact set of integers. Symbol table technology provides an efficient and effective way to implement many of these mappings. The classic examples map a textual string, such as the name of a variable or temporary, into an integer. Key considerations that arise in symbol table implementation include scalability, space efficiency, and cost of creation, insertion, deletion, and destruction, both for individual entries and for new scopes.

This section presented a simple and intuitive approach to implementing a symbol table: linked sheafs of hash tables. (Section B.4, in Appendix B, presents several alternative implementation schemes.) In practice, this simple scheme works well in many applications inside a compiler, ranging from the parser's symbol table to tracking information for superlocal value numbering (see Section 8.5.1).

Review Questions

1. Using the "sheaf-of-tables" scheme, what is the complexity of inserting a new name into the table at the current scope? What is the complexity of looking up a name declared at an arbitrary scope? What is, in your experience, the maximum lexical-scope nesting level for programs that you write?

2. When the compiler initializes a scope, it may need to provide an initial symbol table size. How might you estimate that initial symbol table size in the parser? How might you estimate it in subsequent passes of the compiler?

5.6 SUMMARY AND PERSPECTIVE

The choice of an intermediate representation has a major impact on the design, implementation, speed, and effectiveness of a compiler. None of the intermediate forms described in this chapter are, definitively, the right answer for all compilers or all tasks in a given compiler. The designer must consider the overall goals of a compiler project when selecting an intermediate form, designing its implementation, and adding auxiliary data structures such as symbol and label tables.

Contemporary compiler systems use all manner of intermediate representations, ranging from parse trees and abstract syntax trees (often used in source-to-source systems) through lower-than-machine-level linear codes (used, for example, in the Gnu compiler systems). Many compilers use multiple IRS—building a second or third one to perform a particular analysis or transformation, then modifying the original, and definitive, one to reflect the result.

■ CHAPTER NOTES

The literature on intermediate representations and experience with them is sparse. This is somewhat surprising because of the major impact that decisions about IRS have on the structure and behavior of a compiler. The classic IR forms have been described in a number of textbooks [7, 33, 147, 171]. Newer forms like SSA [50, 110, 270] are described in the literature on analysis and optimization. Muchnick provides a modern treatment of the subject and highlights the use of multiple levels of IR in a single compiler [270].

The idea of using a hash function to recognize textually identical operations dates back to Ershov [139]. Its specific application in Lisp systems seems to appear in the early 1970s [124, 164]; by 1980, it was common enough that McCarthy mentions it without citation [259].

Cai and Paige introduced multiset discrimination as an alternative to hashing [65]. Their intent was to provide an efficient lookup mechanism with guaranteed constant time behavior. Note that closure-free regular expressions, described in Section 2.6.3, can be applied to achieve a similar effect. The work on shrinking the size of $\mathcal{R}^n$'s AST was done by David Schwartz and Scott Warren.

In practice, the design and implementation of an IR has an inordinately large impact on the eventual characteristics of the completed compiler. Large, complex IRs seem to shape systems in their own image. For example, the large ASTs used in early 1980s programming environments like $\mathcal{R}^n$ limited the size of programs that could be analyzed. The RTL form used in GCC has a low level of abstraction. Accordingly, the compiler does a fine job of managing details such as those needed for code generation, but has few, if any, transformations that require source-like knowledge, such as loop blocking to improve memory hierarchy behavior.

EXERCISES

Section 5.2

1. A parse tree contains much more information than an abstract syntax tree.
   a. In what circumstances might you need information that is found in the parse tree but not the abstract syntax tree?
   b. What is the relationship between the size of the input program and its parse tree? Its abstract syntax tree?
   c. Propose an algorithm to recover a program's parse tree from its abstract syntax tree.

2. Write an algorithm to convert an expression tree into a DAG.

Section 5.3

3. Show how the following code fragment

```
if (c[i] ≠ 0)
   then a[i] ← b[i] ÷ c[i];
   else a[i] ← b[i];
```

   might be represented in an abstract syntax tree, in a control-flow graph, and in quadruples. Discuss the advantages of each representation. For what applications would one representation be preferable to the others?

4. Examine the code fragment shown in Figure 5.13. Draw its CFG and show its SSA form as a linear code.

5. Show how the expression x - 2×y might be translated into an abstract syntax tree, one-address code, two-address code, and three-address code.

6. Given a linear list of ILOC operations, develop an algorithm that finds the basic blocks in the ILOC code. Extend your algorithm to build a control-flow graph to represent the connections between blocks.

Section 5.4

7. For the code shown in Figure 5.14, find the basic blocks and construct the CFG.

```
...
x ← ...
y ← ...
a ← y + 2
b ← 0
while(x < a)
   if (y < x)
      x ← y + 1
      y ← b × 2
   else
      x ← y + 2
      y ← a ÷ 2;
   w ← x + 2
   z ← y × a
   y ← y + 1
```

■ **FIGURE 5.13** Code Fragment for Exercise 4.

```
L01: add    ra,rb   ⇒ r1          L05: add    r9,rb    ⇒ r11
     add    rc,rd   ⇒ r2               add    ra,rb    ⇒ r12
     add    r1,r2   ⇒ r3               add    rc,rd    ⇒ r13
     add    ra,rb   ⇒ r4               i2i    ra       ⇒ r13
     cmp_LT r1,r2   ⇒ r5               add    r13,rb   ⇒ r14
     cbr    r5      → L02,L04          multI  r12,17   ⇒ r15
L02: add    ra,rb   ⇒ r6               jumpI           → L03
     multI  r6,17   ⇒ r7          L06: add    r1,r2    ⇒ r16
     jumpI          → L03              i2i    r2       ⇒ r17
L03: add    ra, rb  ⇒ r22              i2i    r1       ⇒ r18
     multI  r22,17  ⇒ r23              add    r17,r18  ⇒ r19
     jumpI          → L07              add    r18,r17  ⇒ r20
L04: add    rc,rd   ⇒ r8               multI  r1,17    ⇒ r21
     i2i    ra      ⇒ r9               jumpI           → L03
     cmp_LT r9,rd   ⇒ r10         L07: nop
     cbr    r10     → L05,L06
```

■ **FIGURE 5.14** Code Fragment for Exercise 7.

8. Consider the three C procedures shown in Figure 5.15.

a. Suppose a compiler uses a register-to-register memory model. Which variables in procedures A, B, and C would the compiler be forced to store in memory? Justify your answers.

b. Suppose a compiler uses a memory-to-memory model. Consider the execution of the two statements that are in the if clause of the

```
static int max = 0;

void A(int b, int e)
{
  int a, c, d, p;

  a = B(b);
  if (b > 100) {
     c = a + b;
     d = c * 5 + e;
  }
  else
     c = a * b;

  p = c;
  C(&p);
}
```

```
int B(int k)
{
  int x, y;

  x = pow(2, k);
  y = x * 5;
  return y;
}


void C(int *p)
{
  if (*p > max)
     max = *p;
}
```

■ **FIGURE 5.15** Code for Exercise 8.

if-else construct. If the compiler has two registers available at that point in the computation, how many loads and stores would the compiler need to issue in order to load values in registers and store them back to memory during execution of those two statements? What if the compiler has three registers available?

9. In FORTRAN, two variables can be forced to begin at the same storage location with an equivalence statement. For example, the following statement forces a and b to share storage:

```
equivalence (a,b)
```

Can the compiler keep a local variable in a register throughout the procedure if that variable appears in an equivalence statement? Justify your answer.

Section 5.5

10. Some part of the compiler must be responsible for entering each identifier into the symbol table.

a. Should the scanner or the parser enter identifiers into the symbol table? Each has an opportunity to do so.

b. Is there an interaction between this issue, declare-before-use rules, and disambiguation of subscripts from function calls in a language with the FORTRAN 77 ambiguity?

11. The compiler must store information in the IR version of the program that allows it to get back to the symbol table entry for each name. Among the options open to the compiler writer are pointers to the

```
 1  procedure main
 2     integer a, b, c;
 3     procedure f1(w,x);
 4        integer a,x,y;
 5        call f2(w,x);
 6        end;
 7     procedure f2(y,z)
 8        integer a,y,z;
 9        procedure f3(m,n)
10           integer b, m, n;
11           c = a * b * m * n;
12           end;
13        call f3(c,z);
14        end;
15     ...
16     call f1(a,b);
17     end;
```

■ **FIGURE 5.16** Program for Exercise 12.

original character strings and subscripts into the symbol table. Of course, the clever implementor may discover other options.
What are the advantages and disadvantages of each of these representations for a name? How would you represent the name?

12. You are writing a compiler for a simple lexically-scoped language. Consider the example program shown in Figure 5.16.

a. Draw the symbol table and its contents at line 11.

b. What actions are required for symbol table management when the parser enters a new procedure and when it exits a procedure?

13. The most common implementation technique for a symbol table uses a hash table, where insertion and deletion are expected to have **O**(1) cost.

a. What is the worst-case cost for insertion and for deletion in a hash table?

b. Suggest an alternative implementation scheme that guarantees **O**(1) insertion and deletion.