Appendix B

Data Structures

CHAPTER OVERVIEW

Compilers execute so many times that the compiler writer must pay attention to the efficiency of each pass in the compiler. Both asymptotic complexity and expected complexity matter. This appendix presents background material on algorithms and data structures used to address problems in different phases of the compiler.

Keywords: Set Representation, Intermediate Representations, Hash Tables, Lexically Scoped Symbol Tables

B.1 INTRODUCTION

Crafting a successful compiler requires attention to many details. This appendix explores some of the algorithmic issues that arise in compiler design and implementation. In most cases, these details would distract from the relevant discussion in the body of the text. We have gathered them together into this appendix, where they can be considered as needed.

This appendix focuses on the infrastructure to support compilation. Many engineering issues arise in the design and implementation of that infrastructure; the manner in which the compiler writer resolves those issues has a large impact on both the speed of the resulting compiler and the ease of extending and maintaining the compiler. As an example of the issues that arise, the compiler cannot know the size of its inputs until it has read them; thus, the front end must be designed to expand the size of its data structures gracefully in order to accommodate large input files. As a corollary, however, the compiler should know the approximate sizes needed for most of its internal data structures when it invokes the passes that follow the front end. Having generated an IR program with 10,000 names, the compiler should not begin its second pass with a symbol table sized for 1024 names. Any file that

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00015-3

contains IR should begin with a specification of the rough sizes of major data structures.

Similarly, the later passes of a compiler can assume that the IR program presented to them was generated by the compiler. While they should do a complete job of error detection, the implementor need not spend as much time explaining errors and trying to correct them as might be expected in the front end. A common strategy is to build a validation pass that performs a thorough check on the IR program and can be inserted for debugging purposes, and to rely on less-strenuous error detection and reporting when not debugging the compiler. Throughout the process, however, the compiler writers should remember that they are the people most likely to look at the code between passes. Effort spent to make the external forms of the IR more readable often reward the very people who invested the time and effort in it.

B.2 REPRESENTING SETS

Many different problems in compilation are formulated in terms that involve sets. They arise at many points in the text, including the subset construction (Chapter 2), the construction of the canonical collection of LR(1) items (Chapter 3), data-flow analysis (Chapters 8 and 9), and worklists such as the ready queue in list scheduling (Chapter 12). In each context, the compiler writer must select an appropriate set representation. In many cases, the efficiency of the algorithm depends on careful selection of a set representation. (For example, the *IDoms* data structure in the dominance computation represents all the dominator sets, as well as the immediate dominators, in one compact array.)

A fundamental difference between building a compiler and building other kinds of systems software—such as an operating system—is that many problems in compilation can be solved offline. For example, the bottom-up local algorithm for register allocation described in Section 13.3.2 was proposed in the mid-1950s for the original FORTRAN compiler. It is better known as Belady's MIN algorithm for offline page replacement, which has long been used as a standard against which to judge the effectiveness of online page-replacement algorithms. In an operating system, the algorithm is of only academic interest because it is an offline algorithm. Since the operating system cannot know what pages will be needed in the future, it cannot use an offline algorithm. On the other hand, the offline algorithm is practical for a compiler because the compiler can look through an entire block before making decisions.

The offline nature of compilation allows the compiler writer to use a broad variety of set representations. Many representations for sets have been explored. In particular, offline computation often lets us restrict the members

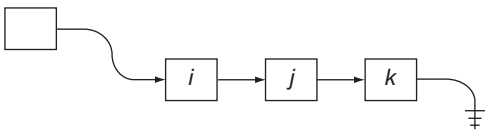
of a set S to a fixed-size universe U ($S \subseteq U$). This, in turn, lets us use more efficient set representations than are available in an online situation where the size of U is discovered dynamically.

Common set operations include `member`, `insert`, `delete`, `clear`, `select`, `cardinality`, `forall`, `copy`, `compare`, `union`, `intersect`, `difference`, and `complement`. A specific application typically uses only a small subset of these operations. The cost of individual set operations depends on the particular representation chosen. In selecting an efficient representation for a particular application, it is important to consider how frequently each type of operation will be used. Other factors to consider include the memory requirements of the set representation and the expected sparsity of S relative to U.

The rest of this section focuses on three efficient set representations that have been employed in compilers: ordered linked lists, bit vectors, and sparse sets.

B.2.1 Representing Sets as Ordered Lists

In cases in which the size of each set is small, it sometimes makes sense to use a simple linked-list representation. For a set S, this representation consists of a linked list and a pointer to the first element in the list. Each node in the list contains a representation for a single element of S and a pointer to the next element of the list. The final node on the list has its pointer set to a standard value indicating the end of the list. With a linked-list representation, the implementation can impose an order on the elements to create an ordered list. For example, an ordered linked list for the set $S = \{i, j, k\}, i < j < k$ might look like this:



The elements are kept in ascending order. The size of S's representation is proportional to the number of elements in S, not the size of U. If $|S|$ is much smaller than $|U|$, the savings from representing just the elements present in S may more than offset the extra cost incurred for a pointer in each element.

The list representation is particularly flexible. Because nothing in the list relies on either the size of U or the size of S, it can be used in situations in which the compiler is discovering U or S or both, such as the live-range-finding portion of a graph-coloring register allocator.

The table in Figure B.1 shows the asymptotic complexities of common set operations using this representation. Most common set operations on ordered

Operation	Ordered Linked List	Bit Vector	Sparse Set
member	$\mathbf{O}(\|S\|)$	$\mathbf{O}(1)$	$\mathbf{O}(1)$
insert	$\mathbf{O}(\|S\|)$	$\mathbf{O}(1)$	$\mathbf{O}(1)$
delete	$\mathbf{O}(\|S\|)$	$\mathbf{O}(1)$	$\mathbf{O}(1)$
clear	$\mathbf{O}(1)$	$\mathbf{O}(\|U\|)$	$\mathbf{O}(1)$
select	$\mathbf{O}(1)$	$\mathbf{O}(\|U\|)$	$\mathbf{O}(1)$
cardinality	$\mathbf{O}(\|S\|)$	$\mathbf{O}(\|U\|)$	$\mathbf{O}(1)$
forall	$\mathbf{O}(\|S\|)$	$\mathbf{O}(\|U\|)$	$\mathbf{O}(\|S\|)$
copy	$\mathbf{O}(\|S\|)$	$\mathbf{O}(\|U\|)$	$\mathbf{O}(\|S\|)$
compare	$\mathbf{O}(\|S\|)$	$\mathbf{O}(\|U\|)$	$\mathbf{O}(\|S\|)$
union	$\mathbf{O}(\|S\|)$	$\mathbf{O}(\|U\|)$	$\mathbf{O}(\|S\|)$
intersect	$\mathbf{O}(\|S\|)$	$\mathbf{O}(\|U\|)$	$\mathbf{O}(\|S\|)$
difference	$\mathbf{O}(\|S\|)$	$\mathbf{O}(\|U\|)$	$\mathbf{O}(\|S\|)$
complement	—	$\mathbf{O}(\|U\|)$	$\mathbf{O}(\|U\|)$

■ **FIGURE B.1** Asymptotic Time Complexities of Set Operations.

linked lists are $\mathbf{O}(|S|)$ because it is necessary to walk the linked lists to perform the operations. If deallocation does not require walking the list to free the nodes for individual elements, as in some garbage-collected systems or an arena-based system, *clear* takes constant time.

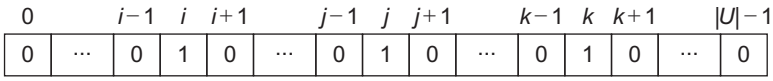
A variant on this idea makes sense when the universe is unknown, and the sets can grow reasonably large, as in interference-graph construction (see Chapter 13). Making each node hold a fixed number (greater than 1) of set elements significantly reduces the overhead in both space and time. With k elements per node, building a set of n elements requires $\lceil \frac{n}{k} \rceil$ allocations and $\lceil \frac{n}{k} \rceil + 1$ pointers, while a set with single-element nodes would take n allocations and $n + 1$ pointers. This scheme retains the easy expansion of the list representation but reduces the space overhead. Insertion and deletion move more data than with a single element per node; however their asymptotic complexity is still $\mathbf{O}(|S|)$.

Keeping the extra space at the front of the list rather than at the end can simplify *insert* and *delete*, assuming a singly linked list.

The *IDoms* array used in the fast dominance computation (see Section 9.5.2) is a clever application of the list representation of sets to a very special case. In particular, the compiler knows the size of the universe and the number of sets. The compiler also knows that, using ordered sets, they will have the peculiar property that if $e \in S_1$ and $e \in S_2$ then every element after e in S_1 is also in S_2. Thus, the elements starting with e can be shared. By using an array representation, the element names can be used as pointers, too. This enables a single array of n elements to represent n sparse sets as ordered lists. It also produces a fast intersection operator for those sets.

B.2.2 Representing Sets as Bit Vectors

Compiler writers often use *bit vectors* to represent sets, particularly those used in data-flow analysis (see Sections 8.6.1 and 9.2). For a bounded universe U, a set $S \subseteq U$ can be represented with a bit vector of length $|U|$, called the *characteristic vector* for S. For each $i \in U$, $0 \leq i < |U|$; if $i \in S$, the i^{th} element of the characteristic vector equals one. Otherwise, the i^{th} element is zero. For example, the characteristic vector for the set $S \subseteq U$, where $S = \{i, j, k\}$, $i < j < k$ is as follows:



The bit-vector representation always allocates enough space to represent all elements in U; thus, this representation can be used only in an application where U is known—an offline application.

The table in Figure B.1 lists the asymptotic complexities of common set operations with this representation. Although many of the operations are $\mathbf{O}(|U|)$, they can still be efficient if U is small. A single word holds many elements; the representation gains a constant-factor improvement over representations that need one word per element. Thus, for example, with a word size of 32 bits, any universe of 32 or fewer elements has a single-word representation.

The compactness of the representation carries over into the speed of operations. With single-word sets, many of the set operations become single machine instructions; for example `union` becomes a logical-or operation and `intersection` becomes a logical-and operation. Even if the sets take multiple words to represent, the number of machine instructions required to perform many of the set operations is reduced by a factor of the machine's word size.

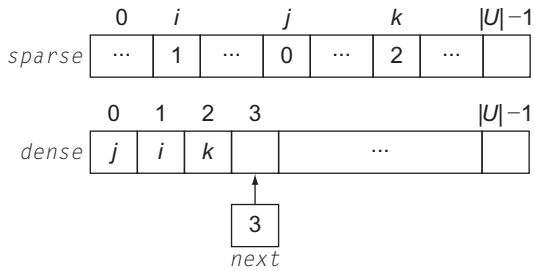
B.2.3 Representing Sparse Sets

For a fixed universe U and a set $S \subseteq U$, S is a sparse set if $|S|$ is much smaller than $|U|$. Some of the sets encountered in compilation are sparse. For example, the LiveOut sets used in register allocation are typically sparse. Compiler writers often use bit vectors to represent such sets, due to their efficiency in time and space. With enough sparsity, however, more time-efficient representations are possible, especially in situations in which a large percentage of the operations can be supported in either $\mathbf{O}(1)$ or $\mathbf{O}(|S|)$ time. By contrast, bit vector sets take either $\mathbf{O}(1)$ or $\mathbf{O}(|U|)$ time on these operations. If $|S|$ is smaller than $|U|$ by a factor greater than the word size, then bit vectors may be the less efficient choice.

One sparse-set representation that has these properties uses two vectors of length $|U|$ and a scalar to represent the set. The first vector, `sparse`, holds a sparse representation of the set; the other vector, `dense`, holds a dense representation of the set. The scalar, `next`, holds the index of the location in `dense` where the next new element of the set can be inserted. Of course, `next` also holds the set's cardinality.

Neither vector needs to be initialized when a sparse set is created; set membership tests ensure the validity of each entry as it is accessed. The `clear` operation simply sets `next` back to zero, its initial value. To add a new element $i \in U$ to S, the code (1) stores i in the `next` location in `dense`, (2) stores the value of `next` in the i^{th} location in `sparse`, and (3) increments `next` so that it is the index of the next location where an element can be inserted in `dense`.

If we began with an empty sparse set S and added the elements j, i, and k, in that order, where $i < j < k$, the set would look like this:



Note that the sparse-set representation requires enough space to represent all of U. Thus, it can be used only in offline situations in which the compiler knows the size of U.

Because valid entries for an element i in `sparse` and `dense` must point to each other, membership can be determined with the following tests:

$$0 \leq sparse[i] < next \quad \text{and} \quad dense[sparse[i]] = i$$

The table in Figure B.1 lists the asymptotic complexities of common set operations. Because this scheme includes both a sparse and a dense representation of the set, it has some of the advantages of each. Individual elements of the set can be accessed in $\mathbf{O}(1)$ time through `sparse`, while set operations that must traverse the set can use `dense` to obtain $\mathbf{O}(|S|)$ complexity.

Both space and time complexities should be considered when choosing between bit-vector and sparse-set representations. The sparse-set representation requires two vectors of length $|U|$ and a scalar. In contrast, a

bit-vector representation requires a single bit-vector of length $|U|$. As shown in Figure B.1, the sparse-set representation dominates the bit-vector representation in terms of asymptotic time complexity. However, because of the efficient implementations possible for bit-vector set operations, bit vectors are preferred in situations where S is not sparse. When choosing between the two representations, it is important to consider the sparsity of the represented set and the relative frequency of the set operations employed.

B.3 IMPLEMENTING INTERMEDIATE REPRESENTATIONS

After choosing a specific style of IR, the compiler writer must decide how to implement it. At first glance, the choices seem obvious. DAGs are easily represented as nodes and edges, using pointers and heap-allocated data structures. Quadruples fall naturally into a $4 \times k$ array. As with sets, however, choosing the best implementation requires a deeper understanding of how the compiler will use the data structures.

B.3.1 Graphical Intermediate Representations

Compilers use a variety of graphical IRs, as discussed in Chapter 5. Tailoring the implementation of a graph to the needs of the compiler can improve both the time and space efficiency of the compiler. This section describes some of the issues that arise with trees and graphs.

Representing Trees

The natural representation for trees, in most languages, is as a collection of nodes connected by pointers. A typical implementation allocates the nodes on demand, as the compiler builds the tree. The tree may include nodes of several sizes—for example, varying the number of children in the node and some of the data fields. Alternatively, the tree might be built with a single kind of node, allocated to fit the largest possible node.

Another way to represent the same tree is as an array of node structures. In this representation, pointers are replaced with integer indices and pointer-based references become standard array and structure references. This implementation forces a one-size-fits-all node, but is otherwise similar to the pointer-based implementation.

Each of these schemes has strengths and weaknesses.

- The pointer scheme handles arbitrarily large ASTs. The array scheme requires code to expand the array when the AST grows beyond its initially allocated size.

- The pointer scheme requires an allocation for each node, while the array scheme simply increments a counter (unless it must expand the array). Techniques, like arena-based allocation (see the sidebar "Arena-Based Allocation," in Chapter 6), can reduce the cost of allocation and reclamation.
- The pointer scheme has locality of reference that depends entirely on the behavior of the allocator at run time. The array technique uses consecutive memory locations. One or the other may be desirable on a particular system.
- The pointer scheme is harder to optimize because of the comparatively poor quality of static analysis on pointer-intensive code. By contrast, many of the optimizations developed for dense linear-algebra codes apply to an array scheme. When the compiler is compiled, these optimizations may produce faster code for the array scheme than for the pointer scheme.
- The pointer scheme may be harder to debug than the array implementation. Programmers seem to find array indices more intuitive than memory addresses.
- The pointer system requires a way to encode pointers if the AST must be written to external media. Presumably, this includes traversing the nodes, following the pointers. The array system uses offsets relative to the start of the array, so no translation is required. On many systems, this can be accomplished with a large, block I/O operation.

There are many other tradeoffs. Each must be evaluated in context.

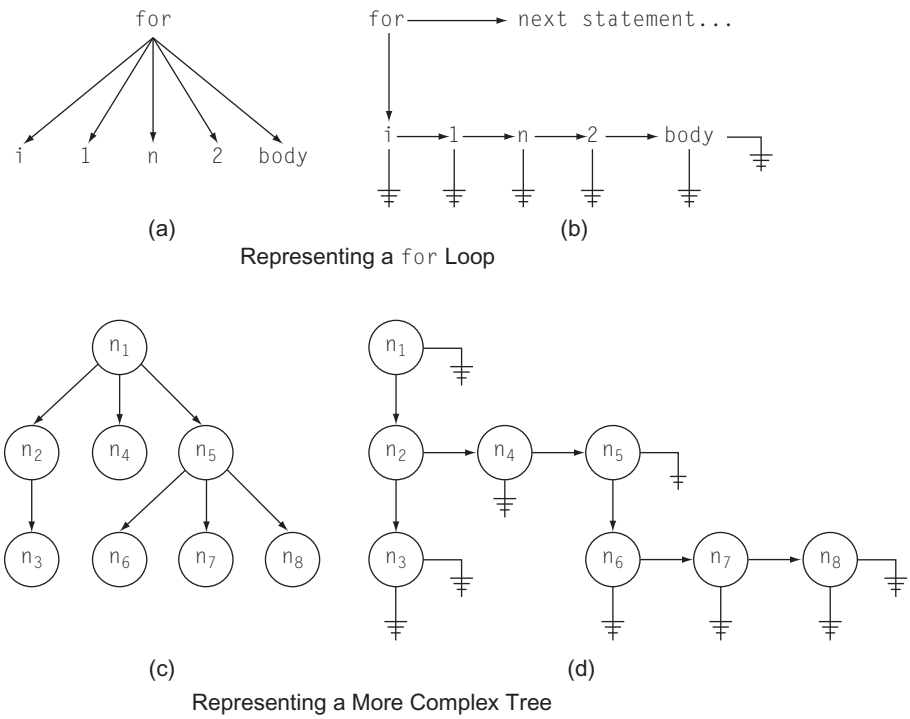
Mapping Trees to Binary Trees

A straightforward implementation of abstract syntax trees might support nodes with many different numbers of children. For example, a typical `for` loop header

```
for i = 1 to n by 2
```

might have a node in the AST with five children, like the one shown in Figure B.2.a. The node labelled `body` represents the subtree for the body of the `for` loop.

For some constructs, no fixed number of children will work. To represent a procedure call, the AST must either custom allocate nodes based on the number of parameters, or use a single child that holds a list of parameters. The former approach complicates all the code that traverses the AST; the variable-sized nodes must hold numbers to indicate how many children they have, and the traversal must contain code to read those numbers and



■ **FIGURE B.2** Mapping Arbitrary Trees onto Binary Trees.

modify its behavior accordingly. The latter approach separates the AST's implementation from its strict adherence to the source but uses a well-understood construct, the list, to represent those places where a fixed-arity node is inappropriate.

To simplify the implementation of trees, the compiler writer can take this separation of form and meaning one step further. Any arbitrary tree can be mapped onto a binary tree—a tree in which each node has precisely two children. In this mapping, the left-child pointer is designated for the leftmost child, and the right-child pointer is designated for the next sibling at the current level. Figure B.2.b shows the five-child `for` node mapped onto a binary tree. Since each node is binary, this tree has null pointers in each leaf node. It also has a sibling pointer in the `for` node; in the version on the left, that pointer occurs in the `for` node's parent. Parts c and d in the figure show a more complex example.

Using binary trees introduces additional null pointers into the trees, as the two examples show. In return, it simplifies the implementation in several ways. Memory allocation can be done simply, with an arena-based allocator

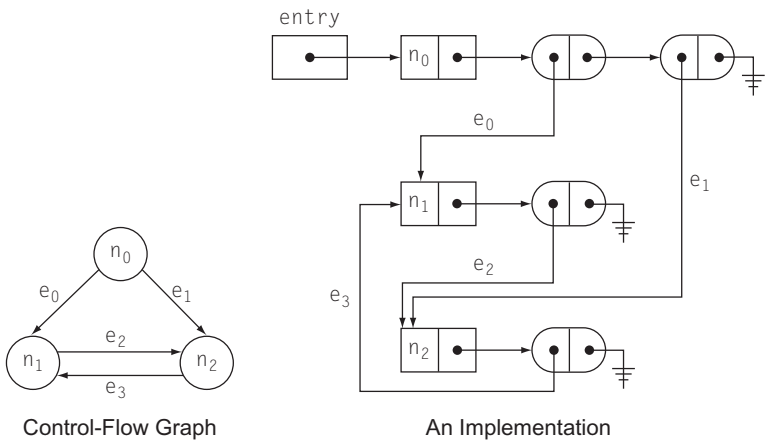
or a custom one. The compiler writer can also implement the tree as an array of structures. The code that deals with the binary tree is somewhat simpler than the code required for a tree with nodes of many different arities.

Representing Arbitrary Graphs

Several structures that a compiler must represent are arbitrary graphs, rather than trees. Examples include the control-flow graph and the data-precedence graph. A simple implementation might use heap-allocated nodes, with pointers to represent the edges. The left side of Figure B.3 shows a simple CFG. Clearly, it needs three nodes. The difficulty arises with the edges: how many incoming and outgoing edges does each node need? Each node could maintain a list of outgoing edges; this leads to an implementation that might look like the one shown on the right side of the figure.

In the diagram, the rectangles represent nodes, and the ovals represent edges. This representation makes it easy to walk the graph in the direction of the edges. It does not provide for random access to any of the nodes; to remedy this, we can add an array of node pointers, indexed by the nodes' integer names. With this minor addition (not shown), the graph is suitable for solving forward data-flow problems. It provides a fast means for finding all the successors of a node.

Unfortunately, compilers often need to traverse the CFG against the direction of the edges. This occurs, for example, in backward data-flow problems, where the algorithm needs a fast predecessor operation. To adapt this graph structure for backward traversal, we would need to add another pointer



■ **FIGURE B.3** An Example Control-Flow Graph.

to each node and create a second set of edge structures to represent the predecessors of a node. This approach will certainly work, but the data structure becomes complicated to draw, implement, and debug.

An alternative, as with trees, is to represent the graph as a pair of tables—one for the nodes and one for the edges. The node table has two fields: one for the first edge to a successor and one for the first edge to a predecessor. The edge table has four fields: the first pair hold the source and sink of the edge being represented, and the other pair holds the next successor of the source and the next predecessor of the sink. Using this scheme, the tables for our example CFG are shown in Figure B.4. This representation provides quick access to successors, predecessors, and individual nodes and edges by their names (assuming that the names are represented by small integers).

The tabular representation works well for traversing the graph and finding predecessors and successors. If the application makes heavy use of other operations on the graph, better representations can be found. For example, the dominant operations in a graph-coloring register allocator are testing for an edge's presence in the interference graph and iterating over a node's neighbors. To support these operations, most implementations use two different graph representations (see Section 13.4.3). To answer membership questions—is the edge (*i, j*) in the graph?—these implementations use a bit matrix. Since the interference graph is undirected, a lower-diagonal bit matrix will suffice, saving roughly half the space required for a full bit matrix. To iterate quickly over a node's neighbors, a set of adjacency vectors is used.

Because interference graphs are both large and sparse, space for the adjacency vectors can become an issue. Some implementations use two passes to build the graph—the first pass computes the size of each adjacency vector and the second pass builds the vectors, each with the minimal required size. Other implementations use a variant of the list representation for sets from Section B.2.1—the graph is built in a single pass, using an unordered list for the adjacency vector, with multiple edges per list node.

Node Table

Name	Successor	Predecessor
n_0	e_0	—
n_1	e_2	e_0
n_2	e_3	e_1

Edge Table

Name	Source	Sink	Next Successor	Next Predecessor
e_0	n_0	n_1	e_1	e_3
e_1	n_0	n_2	—	e_2
e_2	n_1	n_2	—	—
e_3	n_2	n_1	—	—

■ **FIGURE B.4** Tabular Representation of a CFG.

B.3.2 Linear Intermediate Forms

Part of the conceptual appeal of linear intermediate forms, like ILOC, is that they have a simple, obvious implementation as an array of structures. For example, an ILOC program has an immediate mapping to a FORTRAN-style array—n ILOC operations map onto an $(n \times 4)$-element array of integers. The opcode determines how to interpret each of the operands. Of course, any design decision has its advantages and disadvantages, and a compiler writer who wants to use a linear IR should consider representations other than a simple array.

Fortran-Style Array

Using a single array of integers to hold the IR ensures fast access to individual opcodes and operands and low overhead for both allocation and access. The passes that manipulate the IR should run quickly, since all of the array accesses can be improved using the standard analyses and transformations developed to improve dense linear-algebra programs. A linear pass through the code has predictable memory locality; since consecutive operations occupy consecutive memory locations, they cannot conflict in the cache. If the compiler must write the IR to external media (between passes, for example), it can use efficient block I/O operations.

There are, however, disadvantages to the array implementation. If the compiler needs to insert an operation into the code, it must create space for the new operation. Similarly, deletions should contract the code. Any kind of code motion runs into some version of this problem. A naive implementation would create the space by shuffling operations; a compiler that takes this approach will often leave empty slots in the array—after branches and jumps—to reduce the amount of shuffling needed.

An alternative strategy is to use a `detour` operator that directs any traversal of the IR to an out-of-line code segment. This approach lets the compiler thread control through out-of-line segments, so an insertion can be done by overwriting an existing operation with a `detour`, putting the inserted code and the overwritten operation at the end of the array, and following it with a `detour` back to the operation after the first `detour`. The final piece of strategy is to linearize the `detour`s occasionally—for example, at the end of each pass, or any time the fraction of `detour`s exceeds some threshold.

Another complication with the array implementation arises from the need for an occasional operation, such as a ϕ-function that takes a variable number of operands. In the compiler from which our ILOC is derived, procedure calls are represented by a single complicated operation. The call operation has an

operand for each formal parameter, an operand for the return value (if any), and two operands that are lists of values potentially modified by the call and potentially used by the call. This operation does not fit the mold of an $n \times 4$-element array, unless the operands are interpreted as pointers to lists of parameters, modified variables, and used variables.

List of Structures

An alternative to the array implementation is to use a list of structures. In this scheme, each operation has an independent structure, along with a pointer to the next operation. Since the structures can be allocated individually, the program representation expands easily to arbitrary size. Since order is imposed by the pointers that link operations, operations can be inserted or removed with straightforward pointer assignments—no shuffling or copying is required. Variable-length operations, like the call operation previously described, are handled by using variant structures; in fact, short operations such as `loadI` and `jump` can also use a variant to save small amounts of space.

Of course, using individually allocated structures increases the overhead from allocation—the array needed one initial allocation, while the list scheme needs one allocation per IR operation. The list pointers increase the space required. Since all the compiler passes that manipulate the IR must include many pointer-based references, the code for those passes may be slower than code that uses a simple array implementation, because pointer-based code is often harder to analyze and optimize than array-intensive code. Finally, if the compiler writes the IR to external media between passes, it must traverse the list as it writes and reconstruct the list as it reads. This slows down the I/O.

These disadvantages can be ameliorated, to some extent, by implementing the list of structures inside either an arena or an array. With an arena-based allocator, the cost of allocations drops to a test and an addition in the typical case. The arena also produces roughly the same locality as a simple array implementation.

In any pass other than the first one, the compiler should have a fairly accurate notion of how big the IR is. Thus, it can allocate an arena that holds both the IR and some space for growth and avoid the more expensive case of expanding the arena.

Implementing the list in an array achieves the same goals, with the additional advantage that all the pointers become integer indices. Experience suggests that this simplifies debugging; it also makes it possible to use a block I/O operation to write and read the IR.

B.4 IMPLEMENTING HASH TABLES

The two central problems in hash-table implementation are ensuring that the hash function produces an even distribution of integers (at all the table sizes that will be used) and handling collisions in an efficient way. Finding good hash functions is difficult. Fortunately, hashing has been in use long enough that many good functions have been described in the literature.

The rest of this section describes design issues that arise in implementing hash tables. Section B.4.1 describes two hash functions that, in practice, produce good results. The next two sections present the two most widely used strategies for resolving collisions. Section B.4.2 describes *open hashing* (sometimes called *bucket hashing*), while Section B.4.3 presents an alternative scheme called *open addressing* or *rehashing*. Section B.4.4 discusses storage management issues for hash tables, while Section B.4.5 shows how to incorporate the mechanisms for lexical scoping into these schemes. The final section deals with a practical issue that arises in a compiler-development environment, namely, frequent changes to the hash-table definition.

B.4.1 Choosing a Hash Function

The importance of a good hash function cannot be overemphasized. A hash function that produces a bad distribution of index values directly increases the average cost of inserting items into the table and finding such items later. Fortunately, many good hash functions have been documented in the literature, including the multiplicative hash functions described by Knuth and the universal hash functions described by Cormen et al.

Multiplicative Hash Functions

A *multiplicative hash function* is deceptively simple. The programmer chooses a single constant C and uses it in the following formula:

$$h(key) = \lfloor TableSize \cdot ((C \cdot key) \bmod 1) \rfloor$$

where C is the constant, *key* is the integer being used as a key into the table, and *TableSize* is, rather obviously, the current size of the hash table. Knuth suggests the following value for C:

$$0.6180339887 \approx \frac{\sqrt{5}-1}{2}$$

ORGANIZING A SYMBOL TABLE

In designing a symbol table, the first decision that the compiler writer faces concerns the organization of the table and its search algorithm. As in many other applications, the compiler writer has several choices.

Linear List
A linear list can expand to arbitrary size. The search algorithm is a single, small, tight loop. Unfortunately, the search algorithm requires $\mathbf{O}(n)$ probes per lookup, on average, where n is the number of symbols in the table. This single disadvantage almost always outweighs the simplicity of implementation and expansion. To justify using a linear list, the compiler writer needs strong evidence that the procedures being compiled have very few names, as might occur for an object-oriented language.

Binary Search
To retain the easy expansion of the linear list while improving search time, the compiler writer might use a balanced binary tree. Ideally, a balanced tree should allow lookup in $\mathbf{O}(\log_2 n)$ probes per lookup; this is a considerable improvement over the linear list. Many algorithms have been published for balancing search trees. (Similar effects can be achieved by using a binary search of an ordered table, but the table makes insertion and expansion more difficult.)

Hash Table
A hash table may minimize access costs. The implementation computes a table index directly from the name. As long as that computation produces a good distribution of indices, the average access cost should be $\mathbf{O}(1)$. The worst case, however, can devolve to linear search. The compiler writer can take steps to decrease the likelihood of this happening, but pathological cases may still occur. Many hash-table implementations have inexpensive schemes for expansion.

Multiset Discrimination
To avoid worst-case behavior, the compiler writer can use an offline technique called *multiset discrimination*. It creates a distinct index for each identifier, at the cost of an extra pass over the source text. This technique avoids the possibility of pathological behavior that always exists with hashing. (See the sidebar "An Alternative to Hashing," in Chapter 5 for more details.)

Of these organizations, the most common choice appears to be the hash table. It provides better compile-time behavior than the linear list or binary tree, and the implementation techniques have been widely studied and taught.

The effect of the function is to compute $C \cdot key$, take its fractional part with the mod function, and multiply the result by the size of the table.

Universal Hash Functions

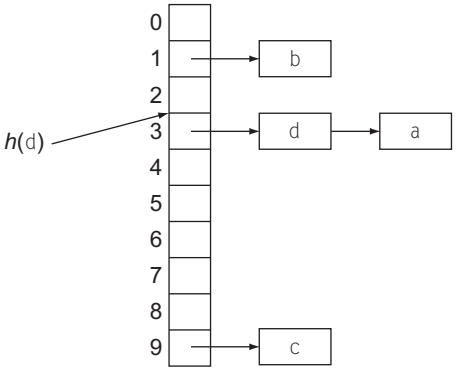
To implement a *universal hash function*, the programmer designs a family of functions that can be parameterized by a small set of constants. At execution time, a set of values for the constants is chosen at random—either using random numbers for the constants or selecting a random index into a set of previously tested constants. (The same constants are used throughout a single execution of the program that uses the hash function, but the constants vary from execution to execution.) By varying the hash function in each execution of the program, a universal hash function produces different distributions in each run of the program. In a compiler, if the input program produced pathological behavior in some particular compilation, it is unlikely to produce the same behavior in subsequent compilations. To implement a universal version of the multiplicative hash function, the compiler writer can randomly generate an appropriate value for C at the start of compilation.

B.4.2 Open Hashing

Open hashing, also called *bucket hashing*, assumes that the hash function h produces collisions. It relies on h to partition the set of input keys into a fixed number of sets, or *buckets*. Each bucket contains a linear list of records, one record per name. `LookUp`(n) walks the linear list stored in the bucket indexed by $h(n)$ to find n. Thus, `LookUp` requires one evaluation of $h(n)$ and the traversal of a linear list. Evaluating $h(n)$ should be fast; the list traversal will take time proportional to the length of the list. For a table of size S, with N names, the cost per lookup should be roughly $\mathbf{O}(\frac{N}{S})$. As long as h distributes names fairly uniformly and the ratio of names to buckets is small, this cost approximates our goal: $\mathbf{O}(1)$ time for each access.

Figure B.5 shows a small hash table implemented with this scheme. It assumes that $h($`a`$) = h($`d`$) = 3$ to create a collision. Thus, `a` and `d` occupy the same slot in the table. The list structure links them together. `Insert` should add to the front of the list for efficiency.

Open hashing has several advantages. Because it creates a new node in one of the linked lists for every inserted name, it can handle an arbitrarily large number of names without running out of space. An excessive number of entries in one bucket does not affect the cost of access in other buckets. Because the concrete representation for the set of buckets is usually an array of pointers, the overhead for increasing S is small—one pointer for each



■ **FIGURE B.5** Open-Hashing Table.

added bucket. (This makes it less expensive to keep $\frac{N}{S}$ small. The cost per name is constant.) Choosing S as a power of two reduces the cost of the inevitable mod operation required to implement h.

The primary drawbacks for open hashing relate directly to these advantages. Both can be managed.

1. Open hashing can be allocation intensive. Each insertion allocates a new record. When implemented on a system with heavy-weight memory allocation, this may be noticeable. Using a lighter-weight mechanism, such as arena-based allocation (see the sidebar in Chapter 6), can alleviate this problem.
2. If any particular set gets large, `LookUp` degrades to linear search. With a reasonably behaved hash function, this occurs only when N is much larger than S. The implementation should detect this problem and enlarge the array of buckets. Typically, this involves allocating a new array of buckets and reinserting each entry from the old table into the new table.

A well-implemented open hash table provides efficient access with low overhead in both space and time.

To improve the behavior of the linear search performed in a single bucket, the compiler can dynamically reorder the chain. Rivest and others [302, 317] describe two effective strategies: move a node up the chain by one position on each lookup, or move it to the front of the list on each lookup. More complex schemes to organize each bucket can be used as well. However, the compiler writer should assess the total amount of time lost in traversing a bucket before investing much effort in this problem.

THE PERILS OF POOR HASH FUNCTIONS

The choice of a hash function has a critical impact on the cost of table insertions and lookups. This is a case in which a small amount of attention can make a large difference.

Many years ago, we saw a student implement the following hash function for character strings: (1) break the key into 4-byte chunks, (2) exclusive-or them together, and (3) take the resulting number, *e*, modulo the table size, as the index. The function is relatively fast. It has a straightforward, efficient implementation. For some table sizes, it produces adequate distributions.

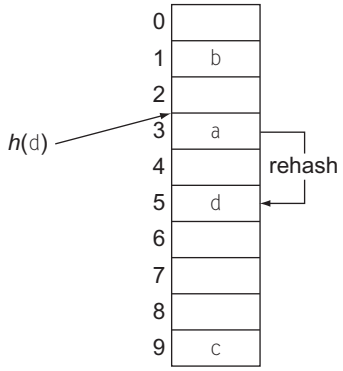
When the student inserted this implementation into a system that performed source-to-source translation on FORTRAN programs, several independent facts combined to create an algorithmic disaster. First, the implementation language padded character strings with blanks to the right to reach a 4-byte boundary. Second, the student chose an initial table size of 2048. Finally, FORTRAN programmers use many one- and two-character variable names, such as `i`, `j`, `k`, `x`, `y`, and `z`.

All the short variable names fit in a single word, avoiding any effect from the exclusive-or. However, taking *e* mod 2048 masks out all but the final 11 bits of *e*. Thus, all short variable names produce the same index—the last 11 bits of a pair of blanks. The hash search instantly devolves into linear search. While this particular hash function is far from ideal, simply changing the table size to 2047 eliminates the most noticeable negative effects.

B.4.3 Open Addressing

Open addressing, also called *rehashing*, handles collisions by computing an alternative index for the names whose normal slot, at $h(n)$, is already occupied. In this scheme, `LookUp`(n) computes $h(n)$ and examines that slot. If the slot is empty, `LookUp` fails. If `LookUp` finds n, it succeeds. If it finds a name other than n, it uses a second function $g(n)$ to compute an increment for the search. This leads it to probe the table at $(h(n) + g(n)) \bmod S$, then at $(h(n) + 2 \times g(n)) \bmod S$, then at $(h(n) + 3 \times g(n)) \bmod S$, and so on, until it either finds n, finds an empty slot, or returns to $h(n)$ a second time. (The table is numbered from 0 to $S - 1$, which ensures that mod S will return a valid table index.) If `LookUp` finds an empty slot, or it returns to $h(n)$ a second time, it fails.

Figure B.6 shows a small hash table implemented with this scheme. It uses the same data as Figure B.5. As before, $h($`a`$) = h($`d`$) = 3$, while $h($`b`$) = 1$ and $h($`c`$) = 9$. When `d` was inserted, it produced a collision with `a`. The secondary hash function $g($`d`$)$ produced 2, so `Insert` placed `d` at index 5 in the table. In effect, open addressing builds chains of items similar to those used in



■ **FIGURE B.6** Open-Addressing Table.

open hashing. In open addressing, however, the chains are stored directly in the table, and a single table location can serve as the starting point for multiple chains, each with a different increment produced by g.

This scheme makes a subtle tradeoff of space against speed. Since each key is stored in the table, S must be larger than N. If collisions are infrequent, because h and g produce good distributions, then the rehash chains stay short and access costs stay low. Because it can recompute g inexpensively, this scheme need not store pointers to form the rehash chains—a savings of N pointers. This saved space goes into making the table larger, and the larger table improves performance by lowering the collision frequency. The primary advantage of open addressing is simple: lower access costs through shorter rehash chains.

Open addressing has two primary drawbacks. Both arise as N approaches S and the table becomes full.

1. Because rehash chains thread through the index table, a collision between n and m can interfere with a subsequent insertion of some other name p. If $h(n) = h(m)$ and $(h(m) + g(m)) \mod S = h(p)$, then inserting n, followed by m, fills p's slot in the table. When the scheme behaves well, this problem has a minor impact. As N approaches S, it can become pronounced.
2. Because S must be at least as large as N, the table must be expanded if N grows too large. (Similarly, the implementation may expand S when some chain becomes too long.) Expansion is needed for correctness; with open hashing, it is a matter of efficiency.

Some implementations use a constant function for g. This simplifies the implementation and reduces the cost of computing secondary indices. However, it creates a single rehash chain for each value of h and has the

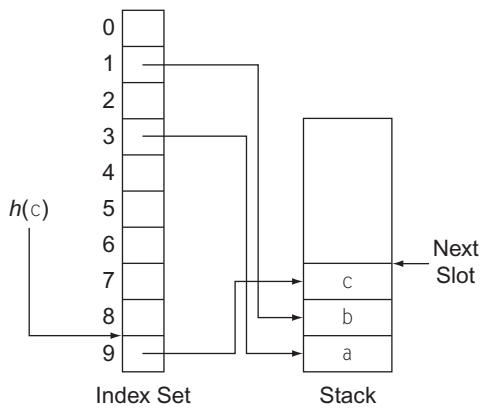
effect of merging rehash chains whenever a secondary index encounters an already occupied table slot. These two disadvantages outweigh the cost of evaluating a second hash function. A more reasonable choice is to use two multiplicative hash functions with different constants, selected randomly at startup from a table of constants, if possible.

The table size S plays an important role in open addressing. `LookUp` must recognize when it reaches a table slot that it has already visited; otherwise, it will not halt on failure. To make this efficient, the implementation should ensure that it eventually returns to $h(n)$. If S is a prime number, then any choice of $0 < g(n) < S$ generates a series of probes, $p_1, p_2, \ldots, p_S$ with the property that $p_1 = p_S = h(n)$ and $p_i \neq h(n), \forall i$ such that $1 < i < S$. That is, `LookUp` will examine every slot in the table before it returns to $h(n)$. Since the implementation may need to expand the table, it should include a table of appropriately sized prime numbers. A small set of primes will suffice, due to the realistic limits on both program size and memory available to the compiler.

B.4.4 Storing Symbol Records

Neither open hashing nor open addressing directly addresses the issue of how to allocate space for the information associated with each hash table entry. With open hashing, the temptation is to allocate the records directly in the nodes that implement the chains. With open addressing, the temptation is to avoid pointers and make each entry in the index table be a symbol record. Both these approaches have drawbacks. We may achieve better results by using a separately allocated stack to hold the records.

Figure B.7 depicts this implementation. In an open-hashing implementation, the chain lists themselves can be implemented on the stack. This



■ **FIGURE B.7** Stack Allocation for Records.

lowers the cost of allocating individual records—particularly if allocation is a heavy-weight operation. In an open-addressing implementation, the rehash chains are still implicit in the index set, preserving the space saving that motivated the idea.

When the actual records are stored in a stack, they form a dense table, which is better for external I/O. For heavyweight allocation, this scheme amortizes the cost of a large allocation over many records. With a garbage collector, it decreases the number of objects that must be marked and collected. In either case, having a dense table makes it more efficient to iterate over the symbols in the table—an operation that the compiler uses to perform tasks such as assigning storage locations.

As a final advantage, this scheme drastically simplifies the task of expanding the index set. To expand the index set, the compiler discards the old index set, allocates a larger set, and then reinserts the records into the new table, working from the bottom of the stack to the top. This eliminates the need to have, temporarily, both the old and new table in memory. Iterating over the dense table takes less work, in general, than chasing the pointers to traverse the lists in open hashing. It avoids iterating over empty table slots, as can happen when open addressing expands the index set to keep the chains short.

The compiler need not allocate the entire stack as a single object. Instead, the stack can be implemented as a chain of nodes that each hold k records, for some reasonable k. When a node becomes full, the implementation allocates a new node, adds it to the end of the chain, and continues. This provides the compiler writer with fine-grained control over the tradeoff between allocation cost and wasted space.

B.4.5 Adding Nested Lexical Scopes

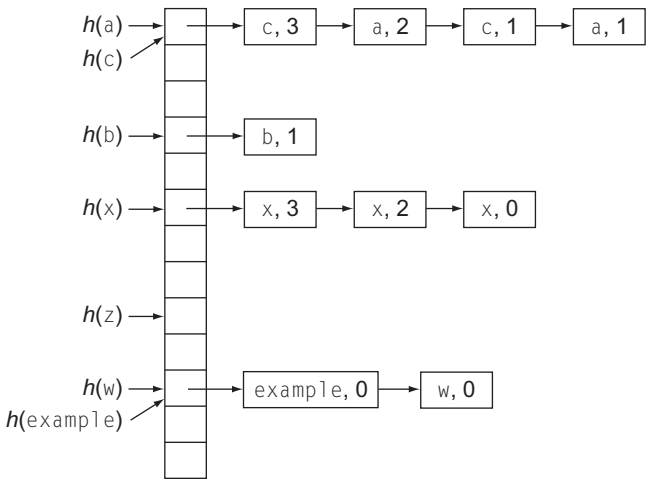
Section 5.5.3 describes the issues that arise in creating a symbol table to handle nested lexical scopes. It describes a simple implementation that creates a sheaf of symbol tables, one per level. While that implementation is conceptually clean, it pushes the overhead of scoping into `LookUp`, rather than into `InitializeScope`, `FinalizeScope`, and `Insert`. Since the compiler invokes `LookUp` many more times than it invokes these other routines, other implementations deserve consideration.

Consider again the code in Figure 5.10. It generates the following actions:

↑ ⟨w,0⟩ ⟨x,0⟩ ⟨example,0⟩ ↑ ⟨a,1⟩ ⟨b,1⟩ ⟨c,1⟩

↑ ⟨b,2⟩ ⟨z,2⟩ ↓ ↑ ⟨a,2⟩ ⟨x,2⟩ ↑ ⟨c,3⟩, ⟨x,3⟩ ↓ ↓ ↓ ↓

where ↑ represents a call to `InitializeScope`, ↓ a call to `FinalizeScope`, and ⟨name, n⟩ represents a call to `Insert` that adds `name` at level n.



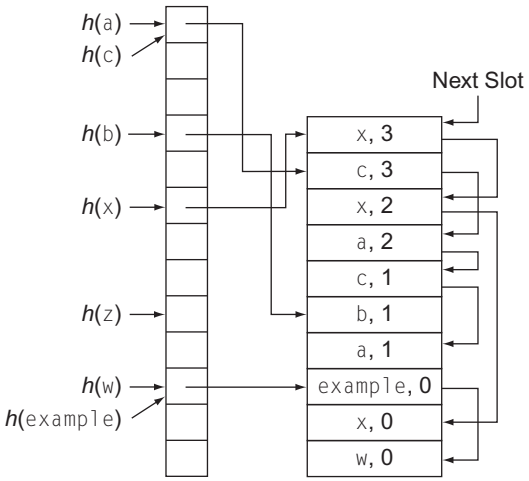
■ **FIGURE B.8** Lexical Scoping in an Open-Hashing Table.

Adding Lexical Scopes to Open Hashing

Consider what might happen in an open-hashing table if we simply add a lexical-level field to the record for each name and insert each new name at the front of its chain. `Insert` could then check for duplicates by comparing both names and lexical levels. `LookUp` would return the first record that it discovered for a given name. `InitializeScope` would simply bump a counter for the current lexical level. This scheme pushes the complications into `FinalizeScope`, which must not only decrement the current lexical level, but also must remove the records for any names inserted in the scope being deallocated.

If open hashing is implemented with individually allocated nodes for its chains, as shown in Figure B.5, then `FinalizeScope` must find all records for the scope being discarded and remove them from their respective chains. If they will not be used later in the compiler, `FinalizeScope` must deallocate them; otherwise, it must chain them together to preserve them. Figure B.8 shows the table that this approach would produce, at the assignment statement in Figure 5.10.

With stack-allocated records, `FinalizeScope` can iterate from the top of the stack downward until it reaches a record for some level below the level being discarded. For each record, it updates the index-set entry with the record's pointer to the next item on the chain. If the records are being discarded, `FinalizeScope` resets the pointer to the next available slot; otherwise, the



■ **FIGURE B.9** Lexical Scoping in a Stack-Allocated Open-Hashing Table.

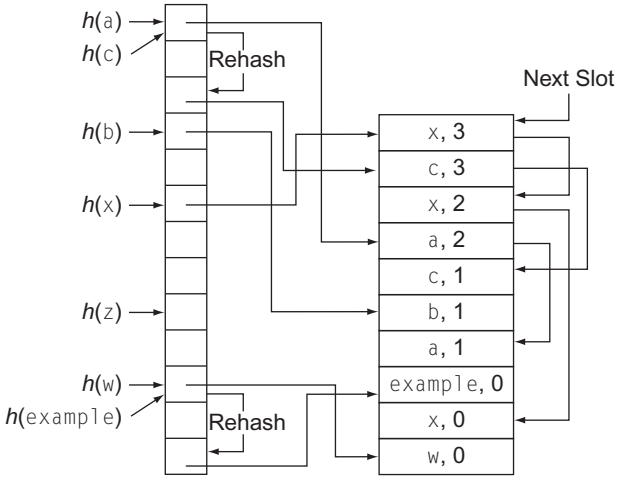
records are preserved together on the stack. Figure B.9 shows the symbol table for our example at the assignment statement.

With a little care, dynamic reordering of the chain can be added to this scheme. Since `FinalizeScope` uses the stack ordering, rather than the chain ordering, it will still find all the top-level names at the top of the stack. With reordered chains, the compiler either needs to walk the chain to remove each deleted name record, or to doubly link the chains to allow quicker deletion.

Adding Lexical Scopes to Open Addressing

With an open-addressing table, the situation is slightly more complex. Slots in the table are a critical resource; when all the slots are filled, the table must be expanded before further insertion can occur. Deletion from a table that uses rehashing is difficult; the implementation cannot easily tell if the deleted record falls in the middle of some rehash chain. Thus, marking the slot empty breaks any chain that passes through that location (rather than ending there). This argues against storing discrete records for each variant of a name in the table. Instead, the compiler should link only one record per name into the table; it can create a chain of superseded records for older variants. Figure B.10 depicts this situation for the continuing example.

This scheme pushes most of the complexity into `Insert` and `FinalizeScope`. `Insert` creates a new record on top of the stack. If it finds an older declaration of the same name in the index set, it replaces that reference with a reference to the new record and links the older reference to the



■ **FIGURE B.10** Lexical Scoping in an Open-Addressing Table.

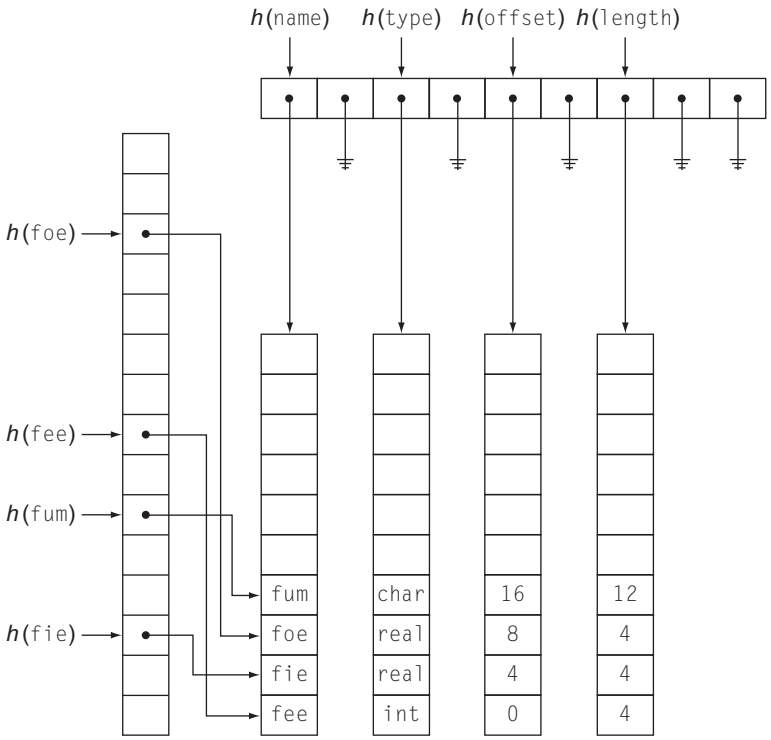
new record. *FinalizeScope* iterates over the top items on the stack, as in open hashing. To remove a record that has an older variant, it simply relinks the index set to point to the older variant. To remove the final variant of a name, it must insert a reference to a specially designated record that denotes a deleted reference. *LookUp* must recognize the deleted reference as occupying a slot in the current chain. *Insert* must know that it can replace a deleted reference with any newly inserted symbol.

This scheme, in essence, creates separate chains for collisions and for redeclarations. Collisions are threaded through the index set. Redeclarations are threaded through the stack. This should reduce the cost of *LookUp* slightly, since it avoids examining more than one record for any single name.

Consider a bucket in open hashing that contains seven declarations for x and a single declaration for y at level zero. *LookUp* might encounter all seven records for x before finding y. With the open-addressing scheme, *LookUp* encounters one record for x and one record for y.

B.5 A FLEXIBLE SYMBOL-TABLE DESIGN

Most compilers use a symbol table as a central repository for information about the various names that arise in the source code, in the IR, and in the generated code. During compiler development, the set of fields in the symbol table seems to grow monotonically. Fields are added to support new passes and to communicate information between passes. When the need for a field disappears, it may or may not be removed from the symbol-table definition.



■ **FIGURE B.11** Two-Dimensional Hashed Symbol Table.

As each field is added, the symbol table swells in size and any parts of the compiler with direct access to the symbol table must be recompiled.

We encountered this problem in the implementation of the $\mathcal{R}^n$ and ParaScope programming environments. The experimental nature of these systems led to a situation where additions and deletions of symbol-table fields were common. To address the problem, we implemented a more complex but more flexible structure for the symbol table—a *two-dimensional hash table*. This eliminated almost all changes to the symbol-table definition and its implementation.

The two-dimensional table, shown in Figure B.11, uses two distinct hash index tables. The first, shown along the left edge of the drawing, corresponds to the sparse index table from Figure B.7. The implementation uses this table to hash on symbol names. The second, shown along the top of the drawing, is a hash table for field names. The programmer references individual fields by both their textual name and the name of the symbol; the implementation hashes the symbol name to obtain an index and the field name to select a

vector of data. The desired attribute is stored in the vector under the symbol's index. It behaves as if each field has its own hash table, implemented as shown in Figure B.7.

While this seems complex, it is not particularly expensive. Each table access requires two hash computations rather than one. The implementation need not allocate storage for a given field until a value is stored in it; this avoids the space overhead of unused fields. It allows individual developers to create and delete symbol-table fields without interfering with other programmers.

Our implementation provided entry points for setting initial values for a field (by name), for deleting a field (by name), and for reporting statistics on field use. This scheme allows individual programmers to manage their own symbol-table use in a responsible and independent way, without interference from other programmers and their code.

As a final issue, the implementation should be abstract with respect to a specific symbol table. That is, it should always take a table instance as a parameter. This allows the compiler to reuse the implementation in many cases, such as the superlocal or dominator-based value numbering algorithms in Chapter 8.

APPENDIX NOTES

Many of the algorithms in a compiler manipulate sets, maps, tables, and graphs. The underlying implementations directly affect the space and time that those algorithms require and, ultimately, the usability of the compiler itself [57]. Algorithms and data-structure textbooks cover many of the issues that this appendix brings together [231, 4, 195, 109, 41].

Our research compilers have used almost all the data structures described in this appendix. We have seen performance problems from data-structure growth in several areas.

- Abstract syntax trees, as mentioned in the sidebar in Chapter 5, can grow unreasonably large. The technique of mapping an arbitrary tree onto a binary tree simplifies the implementation and seems to keep overhead low [231].
- The tabular representation of a graph, with lists of successors and predecessors, has been reinvented many times. It works particularly well for CFGs, for which the compiler iterates over both successors and predecessors. We first used this data structure in the PFC system in 1980.
- The sets in data-flow analysis can grow to occupy hundreds of megabytes. Because allocation and deallocation are performance issues at that scale, we routinely use Hanson's arena-based allocator [179].

- The size and sparsity of interference graphs makes them another area that merits careful consideration. We use the ordered-list variant with multiple set elements per node to keep the cost of building the graph low while managing the space overhead [101].

Symbol tables play a central role in the way that compilers store and access information. Much attention has been paid to the organization of these tables. Reorganizing lists [302, 317], balanced search trees [109, 41] and hashing [231, vol. 3] all play a role in making access to these tables efficient. Knuth [231, vol. 3] and Cormen [109] describe the multiplicative hash function in detail.