Chapter 11

Instruction Selection

CHAPTER OVERVIEW

The compiler's front end and optimizer both operate on the code in its IR form. Before the code can execute on a target processor, the IR form of the code must be rewritten into the processor's instruction set. The process of mapping IR operations into target machine operations is called instruction selection.

This chapter introduces two different approaches to instruction selection. The first uses the technology of tree-pattern matching algorithms. The second builds on the classic late-stage transformation, peephole optimization. Both have found widespread use in real compilers.

Keywords: Instruction Selection, Tree-Pattern Matching, Peephole Optimization

11.1 INTRODUCTION

To translate a program from an intermediate representation such as an abstract syntax tree or a low-level linear code into executable form, the compiler must map each IR construct into a corresponding and equivalent construct in the target processor's instruction set. Depending on the relative levels of abstraction in the IR and the target machine's ISA, this translation can involve elaborating details that are hidden in the IR program or it can involve combining multiple IR operations into a single machine instruction. The specific choices that the compiler makes have an impact on the overall efficiency of the compiled code.

The complexity of instruction selection derives from the large number of alternative implementations that a typical ISA provides for even simple operations. In the 1970s, the DEC PDP-11 had a small and compact

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00011-6

instruction set; thus a good compiler such as the BLISS-11 compiler could perform instruction selection with a simple hand-coded pass. As processor ISAs expanded, the number of possible encodings for each program grew unmanageable. This explosion led to systematic approaches for instruction selection, such as those presented in this chapter.

Conceptual Roadmap

Instruction selection, which maps the compiler's IR into the target ISA, is a pattern-matching problem. At its simplest, the compiler could provide a single target ISA sequence for each IR operation. The resulting selector would provide a template-like expansion that would produce correct code. Unfortunately, that code might make poor use of target machine resources. Better approaches consider many possible code sequences for each IR operation and choose the sequence that has the lowest expected cost.

This chapter presents two approaches to instruction selection: one based on tree-pattern matching and one based on peephole optimization. The former approach relies on a high-level tree notation for both the compiler's IR and the target machine's ISA. The latter approach translates the compiler's ISA into a low-level linear IR, systematically improves that IR, and then maps it into the target ISA. Each of these techniques can produce high-quality code that takes into account local context. Each has been incorporated into tools that take a target machine description and produce a working instruction selector.

Overview

Systematic approaches to code generation make it easier to retarget a compiler. The goal of such work is to minimize the effort required to port the compiler to a new processor or system. Ideally, the front end and the optimizer need minimal changes, and much of the back end can be reused as well. This strategy makes good use of the investment in building, debugging, and maintaining the common parts of the compiler.

In practice, a new language often needs some new operations in the IR. The goal, however, is to extend the IR, rather than to reinvent it.

Much of the responsibility for handling diverse targets rests on the instruction selector. A typical compiler uses a common IR for all targets and, to the extent possible, for all source languages. It optimizes the intermediate form based on a set of assumptions that hold true on most, if not all, target machines. Finally, it uses a back end in which the compiler writer has tried to isolate and extract the target-dependent details.

While the scheduler and register allocator need target-dependent information, good design can isolate that knowledge into a concrete description

of the target machine and its ISA. Such a description might include register-set sizes; the number, capabilities, and operation latencies of the functional units; memory alignment restrictions; and the procedure-call convention. The algorithms for scheduling and allocation are then parameterized by those system characteristics and reused across different ISAs and systems.

Thus, the key to retargetability lies in the implementation of the instruction selector. A retargetable instruction selector consists of a pattern-matching engine coupled to a set of tables that encode the needed knowledge about mapping from the IR to the target ISA. The selector consumes the compiler's IR and produces assembly code for the target machine. In such a system, the compiler writer creates a description of the target machine and runs the back-end generator (sometimes called a *code generator*). The back-end generator, in turn, uses the specification to derive the tables needed by the pattern matcher. Like a parser generator, the back-end generator runs offline during compiler development. Thus, we can use algorithms to create the tables that require more time than algorithms typically employed in a compiler.

While the goal is to isolate all machine-dependent code in the instruction selector, scheduler, and register allocator, the reality almost always falls somewhat short of this ideal. Some machine-dependent details creep, unavoidably, into earlier parts of the compiler. For example, the alignment restrictions on activation records may differ among target machines, changing offsets for values stored in activation records (ARs). The compiler may need to represent features such as predicated execution, branch delay slots, and multiword memory operations explicitly if it is to make good use of them. Still, pushing target-dependent details into instruction selection can reduce the number of changes to other parts of the compiler that are needed to port it to a new target processor.

This chapter examines two approaches to automating the construction of instruction selectors. Section 11.3 revisits the simple treewalk scheme from Chapter 7 and uses it as a detailed introduction to the complexities of instruction selection. The following two sections present different ways to apply pattern-matching techniques to transform IR sequences to assembly sequences. The first technique, in Section 11.4, builds on algorithms for matching tree patterns against trees. The second technique, in Section 11.5, builds on ideas from peephole optimization. Both of these methods are description based. The compiler writer writes a description of the target ISA; a tool then constructs a selector for use at compile time. Both methods have been used in successful portable compilers.

SELECTION, SCHEDULING, AND ALLOCATION

The three major processes in the back end are instruction selection, scheduling, and register allocation. All three processes have a direct impact on the quality of the generated code, and they all interact with each other.

Selection directly changes the scheduling process. Selection dictates both the time required for an operation and the functional units on which it can execute. Scheduling might affect instruction selection. If the code generator can implement an IR operation with either of two assembly operations, and those operations use different resources, the code generator might need to understand the final schedule to ensure the best choice.

Selection interacts with register allocation in several ways. If the target processor has a uniform register set, then the instruction selector can assume an unlimited supply of registers and rely on the allocator to insert the loads and stores needed to fit the values into the register set. If, on the other hand, the target machine has rules that restrict register usage, then the selector must pay close attention to specific physical registers. This can complicate selection and predetermine some or all of the allocation decisions. In this situation, the code generator might use a coroutine to perform local register allocation during instruction selection.

Keeping selection, scheduling, and allocation separate—to the extent possible—can simplify implementation and debugging of each process. However, since each of these processes can constrain the others, the compiler writer must take care to avoid adding unnecessary constraints.

11.2 CODE GENERATION

The compiler's back end must solve three problems to generate executable code for a program in IR form. It must convert the IR operations into operations in the target processor's ISA, a process called *instruction selection*, which is the subject of this chapter. It must select an order in which those operations should execute, a process called *instruction scheduling*, which is the subject of Chapter 12. It must determine, at each point in the final code, which values should reside in registers and which values should reside in memory, a process called *register allocation*, which is the subject of Chapter 13. Most compilers handle these three processes separately. These three distinct but related processes are often lumped together in the term "code generation," even though the instruction selector has the primary responsibility for generating target-machine instructions.

Each of these three problems is, on its own, a computationally hard problem. While it is not clear how to define optimal instruction selection, the problem

of generating the fastest code sequence for a CFG with control flow involves a huge number of alternatives. Instruction scheduling is NP-complete for a basic block under most realistic execution models; moving to larger regions of code does not simplify the problem. Register allocation is, in its general form, also NP-complete in procedures with control flow. Most compilers handle these three problems independently.

The level of exposed detail in the IR program matters. An IR with a higher level of abstraction than the ISA requires the instruction selector to supply additional detail. (Mechanical generation of such detail at this late stage in compilation can lead to template-like code with a low level of customization.) An IR with a lower level of abstraction than the ISA allows the selector to tailor its selections accordingly. Compilers that perform little or no optimization generate code directly from the IR produced by the front end.

The complexity of instruction selection arises from the fact that a typical processor provides many distinct ways to perform the same computation. Abstract away, for the moment, the issues of instruction scheduling and register allocation; we will return to them in the next two chapters. If each IR operation had just one implementation on the target machine, the compiler could simply rewrite each IR operation with the equivalent sequence of machine operations. In most contexts, however, a target machine provides multiple ways to implement each IR construct.

Consider, for example, an IR construct that copies a value from one general-purpose register, r_i, to another, r_j. Assume that the target processor uses ILOC as its native instruction set. As we shall see, even ILOC has enough complexity to expose many of the problems of code generation. The obvious implementation of $r_i \rightarrow r_j$ uses `i2i` $r_i \Rightarrow r_j$; such a register-to-register copy is typically one of the least-expensive operations that a processor provides. However, other implementations abound. These include, for example, each of the following operations:

```
addI r_i,0  ⇒ r_j   subI   r_i,0 ⇒ r_j   multI   r_i,1 ⇒ r_j
divI r_i,1  ⇒ r_j   lshiftI r_i,0 ⇒ r_j   rshiftI r_i,0 ⇒ r_j
and  r_i,r_i ⇒ r_j   orI    r_i,0 ⇒ r_j   xorI    r_i,0 ⇒ r_j
```

Still more possibilities exist. If the processor maintains a register whose value is always 0, another set of operations works, using `add`, `sub`, `lshift`, `rshift`, `or`, and `xor`. A larger set of two-operation sequences, including a store followed by a load, also works.

A human programmer would rapidly discount most, if not all, of these alternate sequences. Using `i2i` is simple, fast, and obvious. An automated

process, however, may need to consider all the possibilities and make the appropriate choices. The ability of a specific ISA to accomplish the same effect in multiple ways increases the complexity of instruction selection. For ILOC, the ISA provides only a few, simple, low-level operations for each particular effect. Even so, it supports myriad ways to implement register-to-register copy.

Real processors are more complex than ILOC. They may include higher-level operations and addressing modes that the code generator should consider. While these features allow a skilled programmer or a carefully crafted compiler to create more efficient programs, they also increase the number of choices that the instruction selector confronts—they make the space of potential implementations larger.

Each alternate sequence has its own costs. Most modern machines implement simple operations, such as `i2i`, `add`, and `lshift`, so that they execute in a single cycle. Some operations, like integer multiplication and division, may take longer. The speed of a memory operation depends on many factors, including the detailed current state of the computer's memory system.

In some cases, the actual cost of an operation might depend on context. If, for example, the processor has several functional units, it might be better to perform a register-to-register copy using an operation other than copy that will execute on an underutilized functional unit. If the unit would otherwise be idle, the operation is, effectively, free. Moving it onto the underutilized unit might actually speed up the entire computation. If the code generator must rewrite the copy to a specific operation that executes only on the underutilized unit, this is a selection problem. If the same operation can run on any unit, it is a scheduling problem.

In most cases, the compiler writer wants the back end to produce code that runs quickly. However, other metrics are possible. For example, if the final code will run on a battery-powered device, the compiler might consider the typical energy consumption of each operation. (Individual operations may consume different amounts of energy.) The costs in a compiler that tries to optimize for energy may be radically different than the costs that a speed metric would involve. Processor energy consumption depends heavily on details of the underlying hardware and, thus, may change from one implementation of a processor to another. Similarly, if code space is critical, the compiler writer might assign costs based solely on sequence length. Alternatively, the compiler writer might simply exclude all multioperation sequences that achieve the same effect as a single-operation sequence.

Since a shorter code sequence fetches fewer bytes from RAM, reducing code space may also reduce energy consumption.

To further complicate matters, some ISAs place additional constraints on specific operations. An integer multiply might need to take its operands from a

subrange of the registers. A floating-point operation might need its operands in even-numbered registers. A memory operation might only execute on one of the processor's functional units. A floating-point unit might include an operation that computes the sequence $(r_i \times r_j) + r_k$ more quickly than the individual multiply and add operations. Load-multiple and store-multiple operations might require contiguous registers. The memory system might deliver its best bandwidth and latency for doubleword or quadword loads, rather than singleword loads. Restrictions such as these constrain instruction selection. At the same time, they increase the importance of finding a solution that uses the best operation at each point in the input program.

When the level of abstraction of the IR and the target ISA differ significantly, or the underlying computation models differ, instruction selection can play a critical role in bridging that gap. The extent to which instruction selection can map the computations in the IR program efficiently to the target machine will often determine the efficiency of the generated code. For example, consider three scenarios for generating code from an ILOC-like IR.

1. *A simple, scalar RISC machine* The mapping from IR to assembly is straightforward. The code generator might consider only one or two assembly-language sequences for each IR operation.
2. *A CISC processor* To make effective use of a CISC instruction set, the compiler may need to aggregate several IR operations into one target-machine operation.
3. *A stack machine* The code generator must translate from the register-to-register computational style of ILOC to a stack-based style with its implicit names and, in some cases, destructive operations.

Moving from one-address code to three-address code entails similar problems.

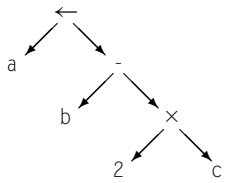
As the gap in abstraction between the IR and the target ISA grows, so does the need for tools to help build code generators.

While instruction selection can play an important role in determining code quality, the compiler writer must keep in mind the enormous size of the search space that the instruction selector might explore. As we shall see, even moderately sized instruction sets can produce search spaces that contain hundreds of millions of states. Clearly, the compiler cannot afford to explore such spaces exhaustively. The techniques that we describe explore the space of alternative code sequences in a disciplined fashion and either limit their searching or precompute enough information to make a deep search efficient.

11.3 EXTENDING THE SIMPLE TREEWALK SCHEME

To make the discussion concrete, consider the issues that can arise in generating code for an assignment statement such as $a \leftarrow b - 2 \times c$. It might be

represented by an abstract syntax tree (AST), as shown on the left, or by a table of quadruples, as shown on the right.



Op	Arg_1	Arg_2	Result
×	2	c	t
-	b	t	a

Instruction selection must produce an assembly-language program from IR representations like these two. For the sake of discussion, assume that it must generate operations in the ILOC subset shown in Figure 11.1.

In Chapter 7, we saw that a simple treewalk routine could generate code from the AST for an expression. The code in Figure 7.5 handled the binary operators, +, -, ×, and ÷ applied to variables and numbers. It generated naive code for the expression and was intended to illustrate an approach that might be used to generate either a low-level, linear IR or assembly code for a simple RISC machine.

The simple treewalk approach generates the same code for every instance of a particular AST node type. While this produces correct code, it never capitalizes on the opportunity to tailor the code to specific circumstances and context. If a compiler performs significant optimization after instruction selection, this may not be a problem. Without subsequent optimization, however, the final code is likely to contain obvious inefficiencies.

Consider, for example, the way that the simple treewalk routine handles variables and numbers. The code for the relevant cases is

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

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

For variables, it relies on two routines, *base* and *offset*, to get the base address and offset into registers. It then emits a `loadAO` operation that adds these two values to produce an effective address and retrieves the contents of the memory location at that address. Because the AST does not differentiate between the storage classes of variables, *base* and *offset* presumably consult the symbol table to obtain the additional information that they need.

CODE LAYOUT

Before it begins emitting code, the compiler has the opportunity to lay out the basic blocks in memory. If each branch in the IR has two explicit branch targets, as ILOC does, then the compiler can choose either of a block's logical successors to follow it in memory. If branches have only one explicit branch target, then rearranging blocks may require rewriting branches—swapping the taken branch and the fall-through branch.

Two architectural considerations should guide this decision. On some processors, taking the branch requires more time than falling through to the next operation. On machines with cache memory, blocks that execute together should be located together. Both of these favor the same strategy for layout. If block *a* ends in a branch that targets *b* and *c*, the compiler should place the more frequently taken target after *a* in memory.

Of course, if a block has multiple predecessors in the control-flow graph, only one of them can immediately precede it in memory. The others will require a branch or jump to reach it (see Section 8.6.2).

Arithmetic Operations		Memory Operations	
add	$r_1, r_2 \Rightarrow r_3$	store	$r_1 \Rightarrow r_2$
addI	$r_1, c_2 \Rightarrow r_3$	storeAO	$r_1 \Rightarrow r_2, r_3$
sub	$r_1, r_2 \Rightarrow r_3$	storeAI	$r_1 \Rightarrow r_2, c_3$
subI	$r_1, c_2 \Rightarrow r_3$	loadI	$c_1 \Rightarrow r_3$
rsubI	$r_2, c_1 \Rightarrow r_3$	load	$r_1 \Rightarrow r_3$
mult	$r_1, r_2 \Rightarrow r_3$	loadAO	$r_1, r_2 \Rightarrow r_3$
multI	$r_1, c_2 \Rightarrow r_3$	loadAI	$r_1, c_2 \Rightarrow r_3$

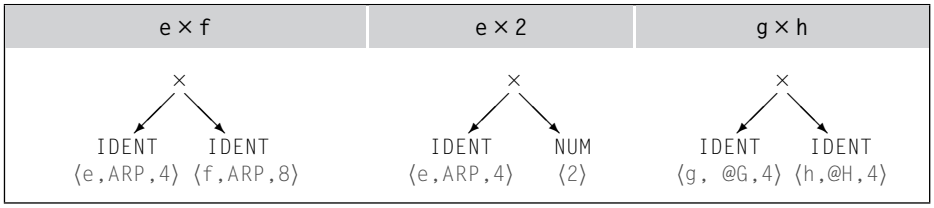
■ **FIGURE 11.1** The ILOC Subset.

Extending this scheme to a more realistic set of cases, including variables that have different-sized representations, call-by-value and call-by-reference parameters, and variables that reside in registers for their entire lifetimes, would require writing explicit code to check all of the cases at each reference. This would make the code for the *IDENT* case much longer (and much slower). It eliminates much of the appealing simplicity of the hand-coded treewalk scheme.

The code to handle numbers is equally naive. It assumes that a number should be loaded into a register in every case, and that *val* can retrieve the number's value from the symbol table. If the operation that uses the number (its parent in the tree) has an immediate form on the target machine and the constant has a value that fits into the immediate field, the compiler should

use the immediate form, since it uses one fewer register. If the number is of a type not supported by an immediate operation, the compiler must arrange to store the value in memory and generate an appropriate memory reference to load the value into a register. This, in turn, may create opportunities for further improvement, such as keeping the constant in a register.

Consider the three multiply operations shown in Figure 11.2. The symbol-table annotations appear below the leaf nodes in the trees. For an identifier, this consists of a name, a label for the base address (or ARP to indicate the current activation record), and an offset from the base address. Below each tree are two code sequences—the code generated by the simple treewalk evaluator and the code we would like the compiler to generate. In the first case, `e × f`, the inefficiency comes from the fact that the treewalk scheme does not generate `loadAI` operations. More complicated code in the *IDENT* case can cure this problem.



Generated Code

```
loadI  4          ⇒ r5
loadAO rarp,r5    ⇒ r6
loadI  8          ⇒ r7
loadAO rarp,r7    ⇒ r8
mult   r6,r8      ⇒ r9
```

```
loadI  4          ⇒ r5
loadAO rarp,r5    ⇒ r6
loadI  2          ⇒ r7
mult   r6,r7      ⇒ r8
```

```
loadI  @G         ⇒ r5
loadI  4          ⇒ r6
loadAO r5,r6      ⇒ r7
loadI  @H         ⇒ r8
loadI  4          ⇒ r9
loadAO r8,r9      ⇒ r10
mult   r7,r10     ⇒ r11
```

Desired Code

```
loadAI r_arp,4    ⇒ r5
loadAI r_arp,8    ⇒ r6
mult   r5,r6      ⇒ r7
```

```
loadAI r_arp,4    ⇒ r5
multI  r5,2       ⇒ r6
```

```
loadI  4          ⇒ r5
loadAI r5,@G      ⇒ r6
loadAI r5,@H      ⇒ r7
mult   r6,r7      ⇒ r8
```

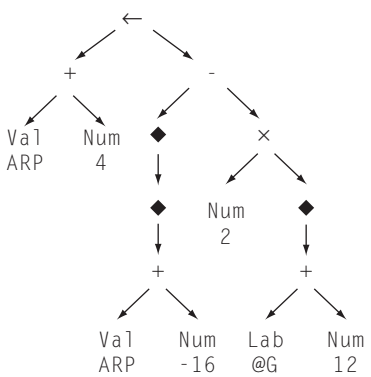
■ **FIGURE 11.2** Variations on Multiply.

The second case, `e × 2`, is harder. The code generator could implement the multiply with a `multI` operation. To recognize this fact, however, the code generator must look beyond the local context. To work this into the treewalk scheme, the case for × might recognize that one subtree evaluates to a constant. Alternatively, the code that handles the *NUM* node might determine that its parent can be implemented with an immediate operation. Either way, it requires nonlocal context that violates the simple treewalk paradigm.

The third case, `g × h`, has another nonlocal problem. Both subtrees of × refer to a variable at offset 4 from its base address. The references have different base addresses. The original treewalk scheme generates an explicit `loadI` operation for each constant—`@G`, `4`, `@H`, and `4`. A version amended to use `loadAI`, as previously mentioned, would either generate separate `loadI`s for `@G` and `@H` or it would generate two `loadI`s for 4. (Of course, the lengths of the values of `@G` and `@H` come into play. If they are too long, then the compiler must use 4 as the immediate operand to the `loadAI` operations.)

The fundamental problem with this third example lies in the fact that the final code contains a common subexpression that was hidden in the AST. To discover the redundancy and handle it appropriately, the code generator would require code that explicitly checks the base address and offset values of subtrees and generates appropriate sequences for all the cases. Handling one case in this fashion would be clumsy. Handling all the similar cases that can arise would require a prohibitive amount of additional coding.

A better way of catching this kind of redundancy is to expose the redundant details in the IR and let the optimizer eliminate them. For the example assignment, `a ← b - 2 × c`, the front end might produce the low-level tree shown in Figure 11.3. This tree has several new kinds of nodes. A `Val` node represents a value known to reside in a register, such as the ARP in r_{arp}.



■ **FIGURE 11.3** Low-Level AST for a ←b - 2 × c.

OPTIMAL CODE GENERATION

The treewalk scheme for selecting instructions produces the same code sequence each time it encounters a particular kind of AST node. More realistic schemes consider multiple patterns and use cost models to choose among them. This leads, naturally, to the question: Can a compiler make optimal choices?

If each operation has an associated cost, and we ignore the effects of instruction scheduling and register allocation, then optimal instruction selection is possible. The tree-pattern-matching code generators described in Section 11.4 produce locally optimal sequences—that is, each subtree is computed by a minimal-cost sequence.

The difficulty of capturing runtime behavior in a single cost number calls into question the importance of such a claim. The impact of execution order, bounded hardware resources, and context-sensitive behavior in the memory hierarchy all complicate the problem of determining the actual cost of any specific code sequence.

In practice, most modern compilers largely ignore scheduling and allocation during instruction selection and assume that the costs associated with various rewrite rules are accurate. Given these assumptions, the compiler looks for locally optimal sequences—those that minimize the estimated cost for an entire subtree. The compiler then performs scheduling and allocation in one or more postpasses over the code produced by instruction selection.

A `Lab` node represents a relocatable symbol, typically an assembly-level label used for either code or data. A ◆ node signifies a level of indirection; its child is an address and it produces the value stored at that address. These new node types require the compiler writer to specify more matching rules. In return, however, additional detail can be optimized, such as the duplicate references to 4 in `g` × `h`.

This version of the tree exposes details at a lower level of abstraction than the target ILOC instruction set. Inspecting this tree reveals, for instance, that `a` is a local variable stored at offset 4 from the ARP, that `b` is a call-by-reference parameter (note the two ◆ nodes), and that `c` is stored at offset 12 from label `@G`. Furthermore, the additions that are implicit in `loadAI` and `storeAI` operations appear explicitly in the tree—as a subtree of a ◆ node or as the left child of an ← node.

Exposing more detail in the AST should lead to better code. Increasing the number of target-machine operations that the code generator considers should also lead to better code. Together, however, these factors create a

situation in which the code generator can discover many different ways to implement a given subtree. The simple treewalk scheme had one option for each AST node type. To make effective use of the target machine's instruction set, the code generator should consider as many possibilities as is practical.

This increased complexity does not arise from a particular methodology or a specific matching algorithm; rather, it reflects a fundamental aspect of the underlying problem—any given machine might provide multiple ways to implement an IR construct. When the code generator considers multiple possible matches for a given subtree, it needs a way to choose among them. If the compiler writer can associate a cost with each pattern, then the matching scheme can select patterns in a way that minimizes the costs. If the costs truly reflect performance, this sort of cost-driven instruction selection should lead to good code.

The compiler writer needs tools that help to manage the complexity of code generation for real machines. Rather than writing code that explicitly navigates the IR and tests the applicability of each operation, the compiler writer should specify rules, and the tools should produce the code required to match those rules with the IR form of the code. The next two sections explore two different approaches to managing the complexity that arises for the instruction set of a modern machine. The next section explores the use of tree-pattern matching techniques. These systems fold the complexity into the process of constructing the matcher, in the same way that scanners fold their choices into the transition tables of DFAs. The following section examines the use of peephole optimization for instruction selection. The peephole-based systems move the complexity of choice into a uniform scheme for low-level simplification followed by pattern matching to find the appropriate instructions. To keep the cost of matching low, these systems limit their scope to short segments of code—two or three operations at a time.

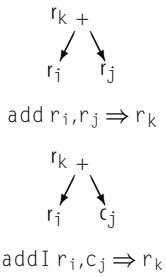
SECTION REVIEW

If the compiler is to take full advantage of the complexities of the target machine, it must expose those complexities in the IR and consider them during instruction selection. Many compilers expand their IR into a detailed low-level form before selecting instructions. Such detailed IRs can be structural, as with our low-level AST, or they can be linear, as we will see in Section 11.5. In either case, the instruction selector must match the details of the IR form of the code to sequences of instructions on the target machine. This section showed that we can expand an ad hoc, treewalk evaluator to perform the task; it also exposed some of the issues that the instruction selector must handle. The next two sections show more general approaches to the problem.

Review Questions

1. To produce the code shown in the right column of Figure 11.2 (on page 606) for the expression `g` × `h`, the instruction selector must differentiate between the length of various constants. For example, the desired code assumes that `@G` and `@H` fit into the immediate field of the `loadAI` operation. How might the IR represent the lengths of these constants? How might the treewalk algorithm account for those lengths?
2. Many compilers use IRs with a higher level of abstraction in the early stages of compilation and then switch to a more detailed IR in the back end. What considerations might argue against exposing low-level details in the early stages of compilation?

11.4 INSTRUCTION SELECTION VIA TREE-PATTERN MATCHING



The compiler writer can use tree-pattern-matching tools to attack the complexity of instruction selection. To transform code generation into tree-pattern matching, both the IR form of the program and the target machine's instruction set must be expressed as trees. As we have seen, the compiler can use a low-level AST as a detailed model of the code being compiled. It can use similar trees to represent the operations available on the target processor. For example, ILOC's addition operations might be modelled by operation trees like those shown in the left margin. By systematically matching operation trees with subtrees of an AST, the compiler can discover all the potential implementations for the subtree.

To work with tree patterns, we need a more convenient notation for describing them. Using a prefix notation, we can write the operation tree for `add` as +(r_i,r_j) and `addI` as +(r_i,c_j). Of course, +(c_i,r_j) is the commutative variant of +(r_i,c_j). The leaves of the operation tree encode information about the storage types of the operands. For example, in +(r_i,c_j), the symbol `r` denotes an operand in a register and the symbol `c` denotes a known constant operand. Subscripts are added to ensure uniqueness, just as we did in the rules for an attribute grammar. If we rewrite the AST from Figure 11.3 in prefix form, it becomes:

```
←(+(Val1,Num1),
   -(◆(◆(+(Val2,Num2))),
   ×(Num3,◆(+(Lab1,Num4)))))
```

While the drawing of the tree may be more intuitive, this linear prefix form contains the same information.

Given an AST and a collection of operation trees, the goal is to map the AST to operations by constructing a *tiling* of the AST with operation trees. A tiling is a collection of ⟨*ast-node*,*op-tree*⟩ pairs, where *ast-node* is a node in the AST and *op-tree* is an operation tree. The presence of an ⟨*ast-node*,*op-tree*⟩ pair in the tiling means that the target-machine operation represented by *op-tree* could implement *ast-node*. Of course, the choice of an implementation for *ast-node* depends on the implementations of its subtrees. The tiling will specify, for each of *ast-node*'s subtrees, an implementation that "connects" with *op-tree*.

A tiling *implements* the AST if it implements every operation and each tile connects with its neighbors. We say that a tile, ⟨*ast-node*,*op-tree*⟩, connects with its neighbors if *ast-node* is covered by a leaf in another *op-tree* in the tiling, unless *ast-node* is the root of the AST. Where two such trees overlap (at *ast-node*), they must agree on the storage class of their common node. For example, if both assume that the common value resides in a register, then the code sequences for the two *op-trees* are compatible. If one assumes that the value resides in memory and the other that it resides in a register, the code sequences are incompatible, since they will not correctly transmit the value from the lower tree to the upper tree.

Given a tiling that implements an AST, the compiler can easily generate assembly code in a bottom-up walk. Thus, the key to making this approach practical lies in algorithms that quickly find good tilings for an AST. Several efficient techniques have emerged for matching tree patterns against low-level ASTs. All these systems associate costs with the operation trees and produce minimal cost tilings. They differ in the technology used for matching—tree matching, text matching, and bottom-up rewrite systems—and in the generality of their cost models—static fixed costs versus costs that can vary during the matching process.

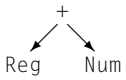
11.4.1 Rewrite Rules

The compiler writer encodes the relationships between operation trees and subtrees in the AST as a set of *rewrite rules*. The rule set includes one or more rules for every kind of node in the AST. A rewrite rule consists of a production in a tree grammar, a code template, and an associated cost. Figure 11.4 shows a set of rewrite rules for tiling our low-level AST with ILOC operations.

Consider rule 16, which corresponds to the tree drawn in the margin. (Its result, at the + node, is implicitly a `Reg`.) The rule describes a

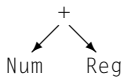
	Production			Cost	Code Template
1	Goal	→	Assign	0	
2	Assign	→	← (Reg_1,Reg_2)	1	store r_2 ⇒ r_1
3	Assign	→	← (+ (Reg_1,Reg_2),Reg_3)	1	storeAO r_3 ⇒ r_1,r_2
4	Assign	→	← (+ (Reg_1,Num_2),Reg_3)	1	storeAI r_3 ⇒ r_1,n_2
5	Assign	→	← (+ (Num_1,Reg_2),Reg_3)	1	storeAI r_3 ⇒ r_2,n_1
6	Reg	→	Lab_1	1	loadI l_1 ⇒ r_{new}
7	Reg	→	Val_1	0	
8	Reg	→	Num_1	1	loadI n_1 ⇒ r_{new}
9	Reg	→	◆ (Reg_1)	1	load r_1 ⇒ r_{new}
10	Reg	→	◆ (+ (Reg_1,Reg_2))	1	loadAO r_1,r_2 ⇒ r_{new}
11	Reg	→	◆ (+ (Reg_1,Num_2))	1	loadAI r_1,n_2 ⇒ r_{new}
12	Reg	→	◆ (+ (Num_1,Reg_2))	1	loadAI r_2,n_1 ⇒ r_{new}
13	Reg	→	◆ (+ (Reg_1,Lab_2))	1	loadAI r_1,l_2 ⇒ r_{new}
14	Reg	→	◆ (+ (Lab_1,Reg_2))	1	loadAI r_2,l_1 ⇒ r_{new}
15	Reg	→	+ (Reg_1,Reg_2)	1	add r_1,r_2 ⇒ r_{new}
16	Reg	→	+ (Reg_1,Num_2)	1	addI r_1,n_2 ⇒ r_{new}
17	Reg	→	+ (Num_1,Reg_2)	1	addI r_2,n_1 ⇒ r_{new}
18	Reg	→	+ (Reg_1,Lab_2)	1	addI r_1,l_2 ⇒ r_{new}
19	Reg	→	+ (Lab_1,Reg_2)	1	addI r_2,l_1 ⇒ r_{new}
20	Reg	→	- (Reg_1,Reg_2)	1	sub r_1,r_2 ⇒ r_{new}
21	Reg	→	- (Reg_1,Num_2)	1	subI r_1,n_2 ⇒ r_{new}
22	Reg	→	- (Num_1,Reg_2)	1	rsubI r_2,n_1 ⇒ r_{new}
23	Reg	→	× (Reg_1,Reg_2)	1	mult r_1,r_2 ⇒ r_{new}
24	Reg	→	× (Reg_1,Num_2)	1	multI r_1,n_2 ⇒ r_{new}
25	Reg	→	× (Num_1,Reg_2)	1	multI r_2,n_1 ⇒ r_{new}

■ **FIGURE 11.4** Rewrite Rules for Tiling the Low-Level Tree with ILOC.



tree that computes the sum of a value located in a Reg and an immediate value in a Num. The left side of the table gives the tree pattern for the rule, Reg → + (Reg_1,Num_2). The center column lists its cost, one. The right column shows an ILOC operation that implements the rule, addI r_1,n_2 ⇒ r_{new}. The operands in the tree pattern, Reg_1 and Num_2, correspond to the operands r_1 and n_2 in the code template. The compiler must rewrite the field r_{new} in the code template with the name of a register allocated to hold the result of the addition. This register name will, in turn, become a leaf in the subtree that connects to this subtree. Notice

that rule 16 has a commutative variant, rule 17. An explicit rule is needed to match subtrees such as the one drawn in the margin.



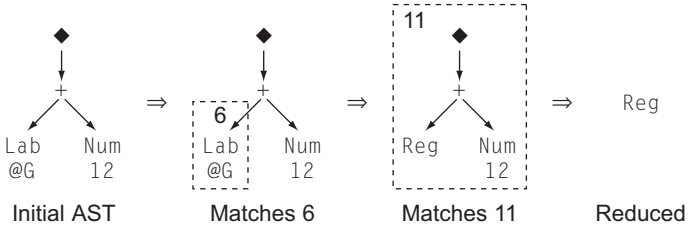
The rules in Figure 11.4 form a tree grammar similar to the grammars that we used to specify the syntax of programming languages. Each rewrite rule, or production, has a nonterminal symbol as its left-hand side. In rule 16, the nonterminal is `Reg`. `Reg` represents a collection of subtrees that the tree grammar can generate, in this case using rules 6 through 25. The right-hand side of a rule is a linearized tree pattern. In rule 16, that pattern is + ($\texttt{Reg}_1$,$\texttt{Num}_2$), representing the addition of two values, a `Reg` and a `Num`.

The rules in Figure 11.4 use `Reg` as both a terminal and a nonterminal symbol in the rules set. This fact reflects an abbreviation in the example. A complete set of rules would include a set of productions that rewrite `Reg` with a specific register name, such as $\texttt{Reg} \rightarrow \texttt{r}_0$, $\texttt{Reg} \rightarrow \texttt{r}_1$, ..., and $\texttt{Reg} \rightarrow \texttt{r}_k$.

The nonterminals in the grammar allow for abstraction. They serve to connect the rules in the grammar. They also encode knowledge about where the corresponding value is stored at runtime and what form it takes. For example, `Reg` represents a value produced by a subtree and stored in a register, while `Val` represents a value already stored in register. A `Val` might be a global value, such as the ARP. It might be the result of a computation performed in a disjoint subtree—a common subexpression.

The cost associated with a production should provide the code generator with a realistic estimate of the runtime cost of executing the code in the template. For rule 16, the cost is one to reflect the fact that the tree can be implemented with a single operation that requires just one cycle to execute. The code generator uses the costs to choose among the possible alternatives. Some matching techniques restrict the costs to numbers. Others allow costs that vary during matching to reflect the impact of previous choices on the cost of the current alternatives.

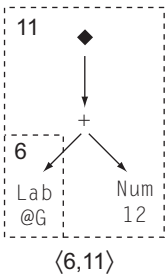
Tree patterns can capture context in a way that the simple treewalk code generator cannot. Rules 10 through 14 each match two operators (◆ and +). These rules express the conditions in which the ILOC operators `loadAO` and `loadAI` can be used. Any subtree that matches one of these five rules can be tiled with a combination of other rules. A subtree that matches rule 10 can also be tiled with the combination of rule 15 to produce an address and rule 9 to load the value. This flexibility makes the set of rewrite rules ambiguous. The ambiguity reflects the fact that the target machine has several ways to implement this particular subtree. Because the treewalk code generator matches one operator at a time, it cannot directly generate either of these ILOC operations.



■ **FIGURE 11.5** A Simple Tree Rewrite Sequence.

To apply these rules to a tree, we look for a sequence of rewriting steps that reduces the tree to a single symbol. For an AST that represents a complete program, that symbol should be the goal symbol. For an interior node, that symbol typically represents the value produced by evaluating the subtree rooted at the expression. The symbol also must specify where the value exists—typically in a register, in a memory location, or as a known constant value.

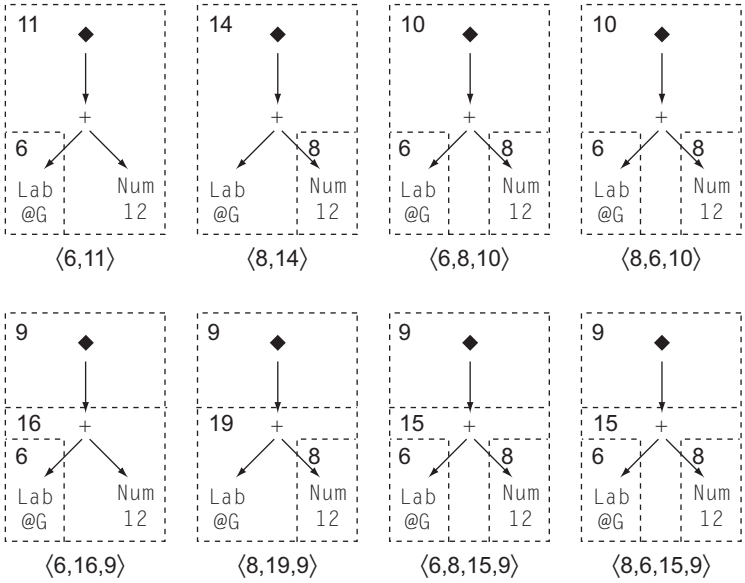
Figure 11.5 shows a rewrite sequence for the subtree that references the variable `c` in Figure 11.3. (Recall that `c` was at offset 12 from the label `@G`.) The leftmost panel shows the original subtree. The remaining panels show one reduction sequence for that subtree. The first match in the sequence recognizes that the left leaf (a `Lab` node) matches rule 6. This allows us to rewrite it as a `Reg`. The rewritten tree now matches the right-hand side of rule 11, ◆ (+ (Reg_1,Num_2)), so we can rewrite the entire subtree rooted at ◆ as a `Reg`. This sequence, denoted ⟨6,11⟩, reduces the entire subtree to a `Reg`.



To summarize such a sequence, we will use a drawing like the one shown in the margin. The dashed boxes show the specific right-hand sides that matched the tree, with the rule number recorded in the upper left corner of each box. The list of rule numbers below the drawing indicates the sequence in which the rules were applied. The rewrite sequence replaces the boxed subtree with the final rule's left-hand side.

Notice how the nonterminals ensure that the operation trees connect appropriately at the points where they overlap. Rule 6 rewrites a `Lab` as a `Reg`. The left leaf in rule 11 is a `Reg`. Viewing the patterns as rules in a grammar folds all of the considerations that arise at the boundaries between operation trees into the labelling of nonterminals.

For this trivial subtree, the rules generate many rewrite sequences, reflecting the ambiguity of the grammar. Figure 11.6 shows eight of these sequences. All the rules in our scheme have a cost of one, except for rules 1 and 7. Since none of the rewrite sequences use these rules, their costs are equal to

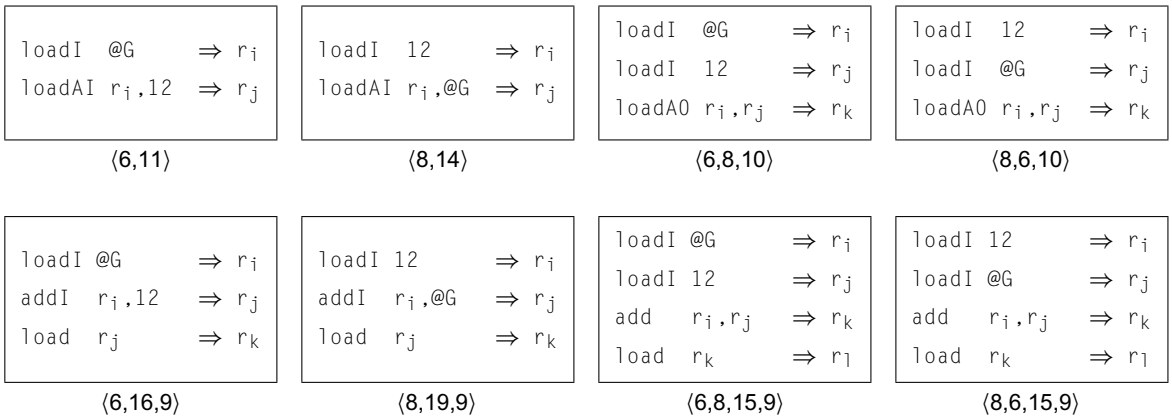


■ **FIGURE 11.6** Potential Matches.

their sequence length. The sequences fall into three categories by cost. The first pair of sequences, ⟨6,11⟩ and ⟨8,14⟩, each have cost two. The next four sequences, ⟨6,8,10⟩, ⟨8,6,10⟩, ⟨6,16,9⟩, and ⟨8,19,9⟩, each have cost three. The final sequences, ⟨6,8,15,9⟩ and ⟨8,6,15,9⟩, each have cost four.

To produce assembly code, the selector uses the code templates associated with each rule. A rule's code template consists of a sequence of assembly-code operations that implements the subtree generated by the production. For example, rule 15 maps the tree pattern + (`Reg`$_1$,`Reg`$_2$) to the code template `add` $r_1, r_2 \Rightarrow r_{new}$. The selector replaces each of r_1 and r_2 with the register name holding the result of the corresponding subtree. It allocates a new virtual register name for r_{new}. A tiling for an AST specifies which rules the code generator should use. The code generator uses the associated templates to generate assembly code in a bottom-up walk. It supplies names, as needed, to tie the storage locations together and emits the instantiated operations corresponding to the walk.

The instruction selector should choose a tiling that produces the lowest-cost assembly-code sequence. Figure 11.7 shows the code that corresponds to each potential tiling. Arbitrary register names have been substituted where appropriate. Both ⟨6,11⟩ and ⟨8,14⟩ produce the lowest cost—two. They lead to different, but equivalent code sequences. Because they have identical



■ **FIGURE 11.7** Code Sequences for the Matches.

costs, the selector is free to choose between them. The other sequences are, as expected, more costly.

If `loadAI` only accepts arguments in a limited range, the sequence ⟨8,14⟩ might not work, since the address that eventually replaces `@G` may be too large for the immediate field in the operation. To handle this kind of restriction, the compiler writer can introduce into the rewriting grammar the notion of a constant with an appropriately limited range of values. It might take the form of a new terminal symbol that can only represent integers in a given range, such as $0 \leq i < 4096$ for a 12-bit field. With such a distinction, and code that checks each instance of an integer to classify it, the code generator could avoid the sequence ⟨8,14⟩, unless `@G` falls in the allowable range for an immediate operand of `loadAI`.

The cost model drives the code generator to select one of the better sequences. For example, notice that the sequence ⟨6,8,10⟩ uses two `loadI` operations, followed by a `loadAO`. The code generator prefers the lower-cost sequences, each of which avoids one of the `loadI` operations and issues fewer operations. Similarly, the cost model avoids the four sequences that use an explicit addition—preferring, instead, to perform the addition implicitly in the addressing hardware.

11.4.2 Finding a Tiling

To apply these ideas to code generation, we need an algorithm that can construct a good tiling, that is, a tiling that produces efficient code. Given a set of rules that encode the operator trees and relate them to the structure of an AST, the code generator should discover an efficient tiling for a specific AST.

Several techniques for constructing such a tiling exist. They are similar in concept, but differ in detail.

To simplify the algorithm, we make two assumptions about the form of the rewrite rules. First, each operation has, at most, two operands. Extending the algorithm to handle the general case is straightforward, but the details complicate the explanation. Second, a rule's right-hand side contains at most one operation. This restriction simplifies the matching algorithm, at no loss in generality. A simple, mechanical procedure can transform the unrestricted case to this simpler case. For a production $\alpha \rightarrow \texttt{op}_1(\beta,\texttt{op}_2(\gamma,\delta))$, rewrite it as $\alpha \rightarrow \texttt{op}_1(\beta,\alpha')$ and $\alpha' \rightarrow \texttt{op}_2(\gamma,\delta)$, where α' is a new symbol that only occurs in these two rules. The resulting growth is linear in the size of the original grammar.

To make this concrete, consider rule 11, `Reg` $\rightarrow$ ◆ `(+(Reg`$_1$`,Num`$_2$`))`. The transformation rewrites it as `Reg` $\rightarrow$ ◆ `(R11P2)` and `R11P2` $\rightarrow$ `+ (Reg`$_1$`,Num`$_2$`)`, where `R11P2` is a new symbol. Notice that the new rule for `R11P2` duplicates rule 16 for `addI`. The transformation adds another ambiguity to the grammar. However, tracking and matching the two rules independently lets the pattern matcher consider the cost of each. The pair of rules that replaces rule 11 should have a cost of one, the cost of the original rule. (Each rule might have fractional cost, or one of them might have zero cost.) This reflects the fact that rewriting with rule 16 produces an `addI` operation, while the rule for `R11P2` folds the addition into the address generation of a `loadAI` operation. The two rule combination, with its lower cost, will guide the pattern matcher to the `loadAI` code sequence when possible—specializing the code to capitalize on the inexpensive addition provided in the `AI` address mode.

The goal of tiling is to label each node in the AST with a set of patterns that the compiler can use to implement it. Since rule numbers correspond directly to right-hand-side patterns, the code generator can use them as a shorthand for the patterns. The compiler can compute sequences of rule numbers, or patterns, for each node in a postorder traversal of the tree. Figure 11.8 sketches an algorithm, *Tile*, that finds tilings for a tree rooted at node *n* in the AST. It annotates each AST node *n* with a set *Label(n)* that contains all the rule numbers that can be used to tile the tree rooted at node *n*. It computes the *Label* sets in a postorder traversal to ensure that it labels a node's children before it labels the node.

Consider the inner loop for the case of a binary node. To compute *Label(n)*, it examines each rule *r* that implements the operation specified by *n*. It uses the functions *left* and *right* to traverse both the AST and the tree patterns (or right-hand sides of the rules). Because *Tile* has already labelled

Each rule specifies an operator and at most two children. Thus, for a rule *r*, *left(r)* and *right(r)* have clear meanings.

```
Tile(n)
  Label(n) ← ∅
  if n is a binary node then
    Tile(left(n))
    Tile(right(n))
    for each rule r that matches n's operation
      if left(r) ∈ Label(left(n)) and right(r) ∈ Label(right(n))
        then Label(n) ← Label(n) ∪ {r}
  else if n is a unary node then
    Tile(left(n))
    for each rule r that matches n's operation
      if left(r) ∈ Label(left(n))
        then Label(n) ← Label(n) ∪ {r}
  else /* n is a leaf */
    Label(n) ← {all rules that match the operation in n}
```

■ **FIGURE 11.8** Compute `Label` Sets to Tile an AST.

`n`'s children, it can use a simple membership test to compare `r`'s children against `n`'s children. If `left(r)` ∈ `Label(left(n))`, then `Tile` has already discovered that it can generate code for `n`'s left subtree in a way that is compatible with using `r` to implement `n`. A similar argument holds for the right subtrees of both `r` and `n`. If both subtrees match, then `r` belongs in `Label(n)`.

A tree-pattern matching code generator built from this algorithm will spend most of its time in the two for loops—computing matches for binary operators or for unary operators. To speed up the code generator, the compiler writer can precompute all the possible matches and store the results in a three-dimensional table, indexed by an operation (`n` in the algorithm) and the label sets of its left and right children. If we replace each of the for loops with a simple table lookup, the algorithm becomes a linear cost walk over the tree.

The tables in this scheme can grow to be large. For example, the lookup table for binary operators has size $|\textit{operation trees}| \times |\textit{label sets}|^2$. The table for unary operators has only two dimensions, with size $|\textit{operation trees}| \times |\textit{label sets}|$. The label sets are bounded in size. If R is the number of rules, then $|\texttt{Label(n)}| \leq R$, and there can be no more than 2^R distinct label sets.

For a machine with 200 operations and a grammar with 1024 distinct label sets ($R = 10$), the resulting table has over 200,000,000 entries. Because the structure of the grammar rules out many possibilities, the tables constructed for this purpose are sparse and can be encoded efficiently. In fact, finding

ways to build and encode these tables efficiently was one of the key advances that made tree-pattern matching a practical tool for code generation.

Finding the Low-Cost Matches

The algorithm in Figure 11.8 finds all of the matches possible within the pattern set. In practice, we want the code generator to find the lowest-cost match. While it could derive the lowest-cost match from the set of all matches, there are more efficient ways to compute the match.

Conceptually, the code generator can discover the lowest-cost match for each subtree in a bottom-up pass over the AST. A bottom-up traversal can compute the cost of each alternative match—the cost of the matched rule plus the costs of the associated subtree matches. In principle, it can discover matches as in Figure 11.8 and retain the lowest-cost ones, rather than all the matches. In practice, the process is slightly more complex.

The cost function depends, inherently, on the target processor; it cannot be derived automatically from the grammar. Instead, it must encode properties of the target machine and reflect the interactions that occur between operations in an assembly program—particularly the flow of values from one operation to another.

A value in the compiled program may have different forms and reside in different locations. For example, a value might reside in a memory location or a register; alternatively, it might be a constant that is small enough to fit into some or all of the immediate operations. (An immediate operand resides in the instruction stream.) Choices among forms and locations matter to the instruction selector because they change the set of target-machine operations that can use the value.

When the instruction selector constructs the set of matches for a particular subtree, it must know the cost of evaluating each of that subtree's operands. If those operands may be in different storage classes—such as registers, memory locations, or immediate constants—the code generator needs to know the cost of evaluating the operand into each of those storage classes. Thus, it must track the lowest-cost sequences that generate each of these storage classes. As it makes the bottom-up traversal to compute costs, the code generator can easily determine the lowest-cost match for each storage class. This adds a small amount of space and time to the process, but the increase is bounded by a factor equal to the number of storage classes—a number that depends entirely on the target machine, and not on the number of rewrite rules.

A careful implementation can accumulate these costs while tiling the tree. If, at each match, the code generator retains the lowest-cost matches, it will

Local optimality
A scheme in which the compiler has no better alternative, at each point in the code, is considered *locally optimal*.

produce a locally optimal tiling. That is, at each node, no better alternative exists, given the rule set and the cost functions. This bottom-up accumulation of costs implements a dynamic-programming solution to finding the minimal-cost tiling.

If we require that the costs be fixed, the cost computation can be folded into the construction of the pattern matcher. This strategy moves computation from compile time into the construction algorithm and almost always produces a faster code generator. If we allow the costs to vary and account for the context in which a match is made, then the cost computation and comparison must be done at compile time. While this scheme may slow down the code generator, it allows more flexibility and precision in the cost functions.

11.4.3 Tools

As we have seen, a tree-oriented, bottom-up approach to code generation can produce efficient instruction selectors. There are several ways that the compiler writer can implement code generators based on these principles.

1. The compiler writer can hand code a matcher, similar to `Tile`, that explicitly checks for matching rules as it tiles the tree. A careful implementation can limit the set of rules that must be examined for each node. This avoids the large sparse table and leads to a compact code generator.
2. Since the problem is finite, the compiler writer can encode it as a finite automaton—a tree-matching automaton—and obtain the low-cost behavior of a DFA. In this scheme, the lookup table encodes the transition function of the automaton, implicitly incorporating all the required state information. Several different systems have been built that use this approach, often called bottom-up rewrite systems (BURS).
3. The grammar-like form of the rules suggests using parsing techniques. The parsing algorithms must be extended to handle the highly ambiguous grammars that result from machine descriptions and to choose least-cost parses.
4. By linearizing the tree into a prefix string, the problem can be translated to a string-matching problem. Then, the compiler can use algorithms from string-pattern matching to find the potential matches.

Tools are available that implement each of the last three approaches. The compiler writer produces a description of a target machine's instruction set, and a code generator creates executable code from the description.

The automated tools differ in details. The cost per emitted instruction varies with the technique. Some are faster, some are slower; none is slow enough that it has a major impact on the speed of the resulting compiler. The

approaches allow different cost models. Some systems restrict the compiler writer to a fixed cost for each rule; in return, they can perform some or all of the dynamic programming during table generation. Others allow more general cost models that may vary the cost during the matching process; these systems must perform the dynamic programming during code generation. In general, however, all these approaches produce code generators that are both efficient and effective.

SECTION REVIEW

Instruction selection via tree-pattern matching relies on the simple fact that trees are a natural representation for both the operations in a program and the operations in the target machine's ISA. The compiler writer develops a library of tree patterns that map constructs in the compiler's IR into operations on the target ISA. Each pattern consists of a small IR tree, a code template, and a cost. The selector finds a low-cost tiling for the tree; in a postorder walk of the tiled tree, it generates code from the templates of the selected tiles.

Several technologies have been used to implement tiling passes. These include hand-coded matchers such as the one shown in Figure 11.8, parser-based matchers operating on ambiguous grammars, linear matchers based on algorithms for fast string matching of the linearized forms, and automata-based matchers. All of these technologies have worked well in one or more systems. The resulting instruction selectors run quickly and produce high-quality code.

Review Questions

1. Tree-pattern matching seems natural for use in a compiler with a tree-like IR. How might sharing in the tree—that is, using a directed acyclic graph (DAG) rather than a tree—affect the algorithm? How might you apply it to a linear IR?
2. Some systems based on tree-pattern matching require that the costs associated with a pattern be fixed, while others allow dynamic costs—costs computed at the time the match is considered. How might the compiler use dynamic costs?

11.5 INSTRUCTION SELECTION VIA PEEPHOLE OPTIMIZATION

Another technique for performing the matching operations that lie at the heart of instruction selection builds on a technology developed for late-stage optimization, called *peephole optimization*. To avoid encoding complexity

in the code generator, this approach combines systematic local optimization on a low-level IR with a simple scheme for matching the IR to target-machine operations. This section introduces peephole optimization, explores its use as a mechanism for instruction selection, and describes the techniques that have been developed to automate construction of peephole optimizers.

11.5.1 Peephole Optimization

The basic premise of peephole optimization is simple: the compiler can efficiently find local improvements by examining short sequences of adjacent operations. As originally proposed, the peephole optimizer ran after all other steps in compilation. It both consumed and produced assembly code. The optimizer had a sliding window, or "peephole," that it moved over the code. At each step, it examined the operations in the window, looking for specific patterns that it could improve. When it recognized a pattern, it would rewrite it with a better instruction sequence. The combination of a limited pattern set and a limited area of focus led to fast processing.

A classic example pattern is a store followed by a load from the same location. The load can be replaced by a copy.

```
storeAI r1     ⇒ rarp,8          storeAI r1 ⇒ rarp,8
loadAI  rarp,8 ⇒ r15      ⇒      i2i     r1 ⇒ r15
```

If the peephole optimizer recognized that this rewrite made the store operation dead (that is, the load was the sole use for the value stored in memory), it could also eliminate the store operation. In general, however, recognizing dead stores requires global analysis that is beyond the scope of a peephole optimizer. Other patterns amenable to improvement by peephole optimization include simple algebraic identities, such as

```
addI r2,0  ⇒ r7          mult r4,r2 ⇒ r10
mult r4,r7 ⇒ r10   ⇒
```

and cases where the target of a branch is, itself, a branch

```
     jumpI → l10          jumpI → l11
l10: jumpI → l11   ⇒   l10: jumpI → l11
```

If this eliminates the last branch to l_{10}, the basic block beginning at l_{10} becomes unreachable and can be eliminated. Unfortunately, proving that the operation at l_{10} is unreachable takes more analysis than is typically available during peephole optimization (see Section 10.2.2).

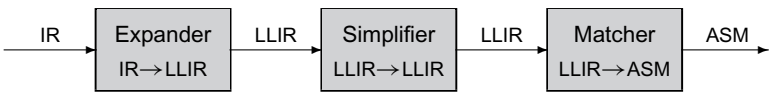
TREE-PATTERN MATCHING ON QUADS?

The terms used to describe these techniques—*tree-pattern matching and peephole optimization*—contain implicit assumptions about the kinds of IR to which they can be applied. BURS theory deals with rewriting operations on trees. This creates the impression that BURS-based code generators require tree-shaped IRs. Similarly, peephole optimizers were first proposed as a final assembly-to-assembly improvement pass. The idea of a moving instruction window strongly suggests a linear, low-level IR for a peephole-based code generator.

Both techniques can be adapted to fit most IRs. A compiler can interpret a low-level linear IR like ILOC as trees. Each operation becomes a tree node; the edges are implied by the reuse of operands. Similarly, if the compiler assigns a name to each node, it can interpret trees as a linear form by performing a postorder treewalk. A clever implementor can adapt the methods presented in this chapter to a wide variety of actual IRs.

Early peephole optimizers used a limited set of hand-coded patterns. They used exhaustive search to match the patterns but ran quickly because of the small number of patterns and the small window size—typically two or three operations.

Peephole optimization has progressed beyond matching a small number of patterns. Increasingly complex ISAs led to more systematic approaches. A modern peephole optimizer breaks the process into three distinct tasks: expansion, simplification, and matching. It replaces the pattern-driven optimization of early systems with a systematic application of symbolic interpretation and simplification.



Structurally, this looks like a compiler. The expander recognizes the input code in IR form and builds an internal representation. The simplifier performs some rewriting operations on that IR. The matcher transforms the IR into target-machine code, typically assembly code (ASM). If the input and output languages are the same, this system is a peephole optimizer. With different languages as input and output, the same algorithms can perform instruction selection, as we shall see in Section 11.5.2.

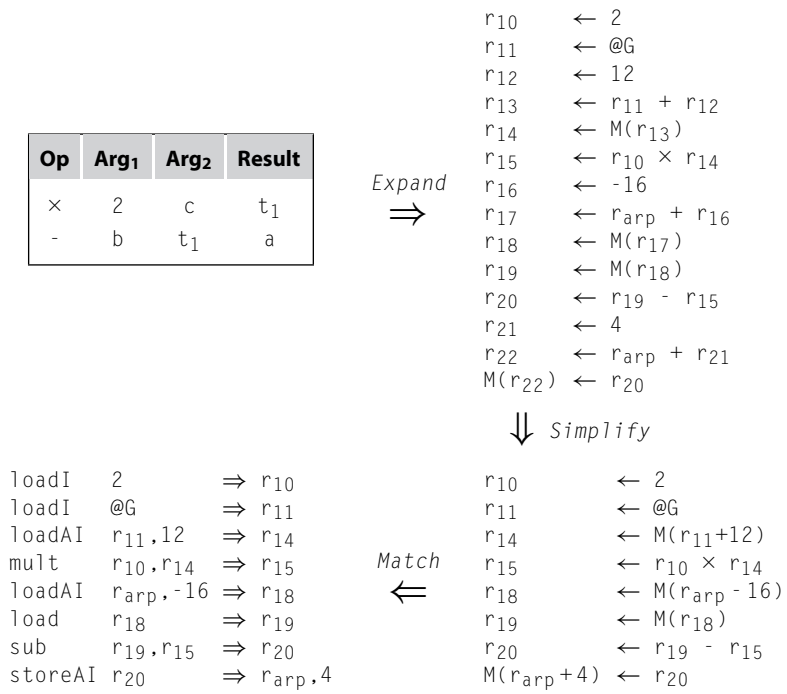
The expander rewrites the IR, operation by operation, into a sequence of lower-level IR (LLIR) operations that represents all the direct effects of an operation—at least, all of those that affect program behavior. If the operation `add` r_i,$r_j \Rightarrow r_k$ sets the condition code, then its LLIR representation must include operations that assign $r_i + r_j$ to r_k and that set the condition code to the appropriate value. Typically, the expander has a simple structure. Operations can be expanded individually, without regard to context. The process uses a template for each IR operation and substitutes appropriate register names, constants, and labels in the templates.

The simplifier makes a pass over the LLIR, examining the operations in a small window on the LLIR and systematically trying to improve them. The basic mechanisms of simplification are forward substitution, algebraic simplification (for example, $x + 0 \Rightarrow x$), evaluating constant-valued expressions (for example, $2 + 17 \Rightarrow 19$), and eliminating useless effects, such as the creation of unused condition codes. Thus, the simplifier performs limited local optimization on the LLIR in the window. This subjects all the details exposed in the LLIR (address arithmetic, branch targets, and so on) to a uniform level of local optimization.

In the final step, the matcher compares the simplified LLIR against the pattern library, looking for the pattern that best captures all the effects in the LLIR. The final code sequence may produce effects beyond those required by the LLIR sequence; for example, it might create a new, albeit useless, condition-code value. It must, however, preserve the effects needed for correctness. It cannot eliminate a live value, regardless of whether the value is stored in memory, in a register, or in an implicitly set location such as the condition code.

Figure 11.9 shows how this approach might work on the example from Section 11.3. It begins, in the upper left, with the quadruples for the low-level AST shown in Figure 11.3. (Recall that the AST computes `a` $\leftarrow$ `b - 2` $\times$ `c`, with `a` stored at offset 4 in the local AR, `b` stored as a call-by-reference parameter whose pointer is stored at offset –16 from the ARP, and `c` at offset `12` from the label `@G`.) The expander creates the LLIR shown on the upper right. The simplifier reduces this code to produce the LLIR code in the bottom right. From this LLIR fragment, the matcher constructs the ILOC code in the lower left.

The key to understanding this process lies in the simplifier. Figure 11.10 shows the successive sequences that the peephole optimizer has in its window as it processes the low-level IR for the example. Assume that it has a three-operation window. Sequence 1 shows the window with the first three operations. No simplification is possible. The optimizer rolls the first

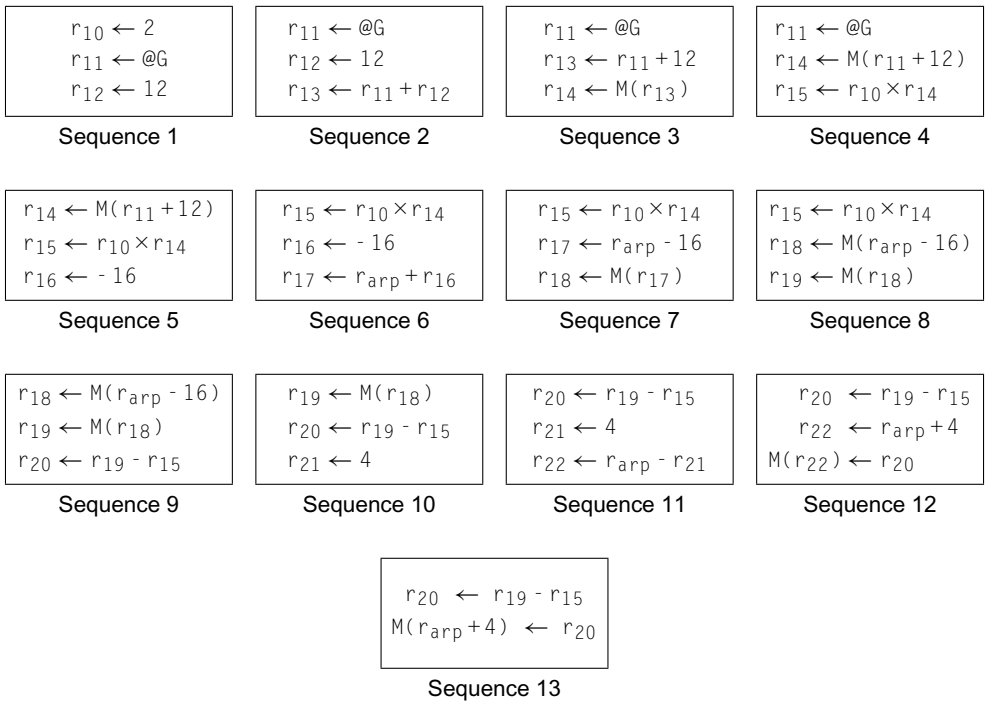


■ **FIGURE 11.9** *Expand*, *Simplify*, and *Match* Applied to the Example.

operation, defining r_{10}, out of the window and brings in the definition of r_{13}. In this window, it can substitute r_{12} forward into the definition of r_{13}. Because this makes r_{12} dead, the optimizer discards the definition of r_{12} and pulls another operation into the bottom of the window to reach sequence 3. Next, it folds r_{13} into the memory reference that defines r_{14}, producing sequence 4.

No simplification is possible on sequence 4, so the optimizer rolls the definition of r_{11} out of the window. It cannot simplify sequence 5, either, so it rolls the definition of r_{14} out of the window, too. It can simplify sequence 6 by forward substituting `-16` into the addition that defines r_{17}. That action produces sequence 7. The optimizer continues in this manner, simplifying the code when possible and advancing when it cannot. When it reaches sequence 13, it halts because it cannot further simplify the sequence and it has no additional code to bring into the window.

Returning to Figure 11.9, compare the simplified code with the original code. The simplified code consists of those operations that roll out the top of the window, plus those left in the window when simplification halts. After



■ **FIGURE 11.10** Sequences Produced by the Simplifier.

simplification, the computation takes 8 operations, instead of 14. It uses 7 registers (other than r_{arp}), instead of 13.

Several design issues affect the ability of a peephole optimizer to improve code. The ability to detect when a value is dead plays a critical role in simplification. The handling of control-flow operations determines what happens at block boundaries. The size of the peephole window limits the optimizer's ability to combine related operations. For example, a larger window would let the simplifier fold the constant 2 into the multiply operation. The next three subsections explore these issues.

Recognizing Dead Values

$r_{12} \leftarrow 2$
$r_{14} \leftarrow r_{12} + r_{12}$

When the simplifier confronts a sequence such as the one shown in the margin, it can fold the value 2 in place of the use of r_{12} in the second operation. It cannot, however, eliminate the first operation unless it knows that r_{12} is not live after the use in the second operation—that is, the value is dead. Thus, the ability to recognize when a value is no longer live plays a critical role in the simplifier's operation.

The compiler can compute LIVEOUT sets for each block and then, in a backward pass over the block, track which values are live at each operation. As an alternative, it can use the insight that underlies the semipruned SSA form; it can identify names that are used in more than one block and consider any such name live on exit from each block. This alternative strategy avoids the expense of live analysis; it will correctly identify any value that is strictly local to the block where it is defined. In practice, the effects introduced by the expander are strictly local so the less expensive approach produces good results.

Given either LIVEOUT sets or the set of global names, the expander can mark last uses in the LLIR. Two observations make this possible. First, the expander can process a block from bottom to top; the expansion is a simple template-driven process. Second, as it walks the block from bottom to top, the expander can build a set of values that are live at each operation, LIVENOW.

Last use
a reference to a name after which the value represented by that name is no longer live

The computation of LIVENOW is simple. The expander sets the initial value for LIVENOW equal to the LIVEOUT set for the block. (In the absence of LIVEOUT sets, it can set LIVENOW to contain all the global names.) Now, as it processes an operation $r_i \leftarrow r_j$ op r_k, the algorithm adds r_j and r_k to LIVENOW and deletes r_i. This algorithm produces, at each step, a LIVENOW set that is as precise as the initial information used at the bottom of the block.

On a machine that uses a condition code to control conditional branches, many operations set the condition code's value. In a typical block, many of those condition code values are dead. The expander must insert explicit assignments to the condition code. The simplifier must understand when the condition code's value is dead because extraneous assignments to the condition code may prevent the matcher from generating some instruction sequences.

For example, consider the computation $r_i \times r_j + r_k$. If both $\times$ and + set the condition code, the two-operation sequence might generate the following LLIR:

$$
\begin{aligned}
r_{t1} &\leftarrow r_i \times r_j \\
cc &\leftarrow f_x(r_i, r_j) \\
\\
r_{t2} &\leftarrow r_{t1} + r_k \\
cc &\leftarrow f_+(r_{t1}, r_k)
\end{aligned}
$$

The first assignment to cc is dead. If the simplifier eliminates that assignment, it can combine the remaining operations into a multiply-add operation, assuming the target machine has such an instruction. If it cannot eliminate

$cc \leftarrow f_X(r_i, r_j)$, however, the matcher cannot use multiply-add because it cannot set the condition code twice.

Control-Flow Operations

The presence of control-flow operations complicates the simplifier. The easiest way to handle them is to clear the simplifier's window when it reaches a branch, a jump, or a labelled instruction. This keeps the simplifier from moving effects onto paths where they were not present.

The simplifier can achieve better results by examining context around branches, but it introduces several special cases to the process. If the input language encodes branches with a single target and a fall-through path, then the simplifier should track and eliminate dead labels. If it eliminates the last use of a label and the preceding block has a fall-through exit, then it can remove the label, combine the blocks, and simplify across the old boundary. If the input language encodes branches with two targets, or the preceding block ends with a jump, then a dead label implies an unreachable block that can be completely eliminated. In either case, the simplifier should track the number of uses for each label and eliminate labels that can no longer be referenced. (The expander can count label references, allowing the simplifier to use a simple reference-counting scheme to track the number of remaining references.)

A more aggressive approach might consider the operations on both sides of a branch. Some simplifications may be possible across the branch, combining effects of the operation immediately before the branch with those of the operation at the branch's target. However, the simplifier must account for *all* the paths reaching the labelled operation.

Predicated operations require some of these same considerations. At runtime, the predicate values determine which operations actually execute. In effect, the predicates specify a path through a simple CFG, albeit one without explicit labels or branches. The simplifier must recognize these effects and treat them in the same cautious fashion that it uses for labelled operations.

Physical versus Logical Windows

The discussion, so far, has focused on a window containing adjacent operations in the low-level IR. This notion has a nice physical intuition and makes the concept concrete. However, adjacent operations in the low-level IR may not operate on the same values. In fact, as target machines offer more instruction-level parallelism, a compiler's front end and optimizer must generate IR programs that have more independent and interleaved computations to keep the target machine's functional units busy. In this case, the peephole optimizer may find very few opportunities for improving the code.

To improve this situation, the peephole optimizer can use a logical window rather than a physical window. With a logical window, it considers operations that are connected by the flow of values within the code—that is, it considers together operations that define and use the same value. This creates the opportunity to combine and simplify related operations, even if they are not adjacent in the code.

During expansion, the optimizer can link each definition with the next use of its value in the block. The simplifier uses these links to fill its window. When the simplifier reaches operation *i*, it constructs a window for *i* by pulling in operations linked to *i*'s result. (Since simplification relies, in large part, on forward substitution, there is little reason to consider the next physical operation, unless it uses *i*'s result.) Using a logical window within a block can make the simplifier more effective, reducing both compile time required and the number of operations remaining after simplification. In our example, a logical window would let the simplifier fold the constant 2 into the multiplication.

Extending this idea to larger scopes adds some complication. The compiler can attempt to simplify operations that are logically adjacent but too far apart to fit in the peephole window together—either within the same block or in different blocks. This requires a global analysis to determine which uses each definition can reach (that is, reaching definitions from Section 9.2.4). Additionally, the simplifier must recognize that a single definition may reach multiple uses, and a single use might refer to values computed by several distinct definitions. Thus, the simplifier cannot simply combine the defining operation with one use and leave the remaining operations stranded. It must either limit its consideration to simple situations, such as a single definition and a single use, or multiple uses with a single definition, or it must perform some careful analysis to determine whether a combination is both safe and profitable. These complications suggest applying a logical window within a local or superlocal context. Moving the logical window beyond an extended basic block adds significant complications to the simplifier.

11.5.2 Peephole Transformers

The advent of more systematic peephole optimizers, as described in the previous section, created the need for more complete pattern sets for a target machine's assembly language. Because the three-step process translates all operations into LLIR and tries to simplify all the LLIR sequences, the matcher needs the ability to translate arbitrary LLIR sequences back into assembly code for the target machine. Thus, these modern peephole systems have much larger pattern libraries than earlier, partial systems. As computers moved from 16-bit instructions to 32-bit instructions, the explosion in the

RISC, CISC, AND INSTRUCTION SELECTION

Early proponents of RISC architectures suggested that RISCs would lead to simpler compilers. Early RISC machines, like the IBM 801, had many fewer addressing modes than contemporary CISC machines (like DEC's VAX-11). They featured register-to-register operations, with separate load and store operations for moving data between registers and memory. In contrast, the VAX-11 accommodated both register and memory operands; many operations were supported in both two-address and three-address forms.

The RISC machines did simplify instruction selection. They offered fewer ways to implement a given operation. They had fewer restrictions on register use. However, their load-store architectures increased the importance of register allocation.

In contrast, CISC machines have operations that encapsulate more complex functionality into a single operation. To make effective use of these operations, the instruction selector must recognize larger patterns over larger code fragments. This increases the importance of systematic instruction selection; the automated techniques described in this chapter are more important for CISC machines, but equally applicable to RISC machines.

number of distinct assembly operations made hand-generation of the patterns problematic. To handle this explosion, most modern peephole systems include a tool that automatically generates a matcher from a description of a target machine's instruction set.

The advent of tools to generate the large pattern libraries needed to describe a processor's instruction set has made peephole optimization a competitive technology for instruction selection. One final twist further simplifies the picture. If the compiler already uses the LLIR for optimization, then the compiler does not need an explicit expander. Similarly, if the compiler optimized the LLIR, the simplifier need not worry about dead effects; it can assume that the optimizer will remove them with its more general techniques for dead-code elimination.

This scheme also reduces the work required to retarget a compiler. To change target processors, the compiler writer must (1) provide an appropriate machine description to the pattern generator so that it can produce a new instruction selector; (2) change the LLIR sequences generated by earlier phases so that they fit the new ISA; and (3) modify the instruction scheduler and register allocator to reflect the characteristics of the new ISA. While this encompasses a significant amount of work, the infrastructure for describing, manipulating, and improving the LLIR sequences remains intact. Put another

way, the LLIR sequences for radically different machines must capture their differences; however, the base language in which those sequences are written remains the same. This allows the compiler writer to build a set of tools that are useful across many architectures and to produce a machine-specific compiler by generating the appropriate low-level IR for the target ISA and providing an appropriate set of patterns for the peephole optimizer.

The other advantage of this scheme lies in the simplifier. This stripped-down peephole transformer still includes a simplifier. Systematic simplification of code, even when performed in a limited window, provides a significant advantage over a simple hand-coded pass that walks the IR and rewrites it into assembly language. Forward substitution, application of simple algebraic identities, and constant folding can produce shorter, more efficient LLIR sequences. These, in turn, may lead to better code for a target machine.

Several important compiler systems have used this approach. The best known may be the Gnu compiler system (GCC). GCC uses a low-level IR known as register-transfer language (RTL) for some of its optimizations and for code generation. The back end uses a peephole scheme to convert RTL into assembly code for target computers. The simplifier is implemented using systematic symbolic interpretation. The matching step in the peephole optimizer actually interprets the RTL code as trees and uses a simple tree-pattern matcher built from a description of the target machine. Other systems, such as Davidson's VPO, construct a grammar from the machine description and generate a small parser that processes the RTL in a linear form to perform the matching step.

SECTION REVIEW

The technology of peephole optimization has been adapted to perform instruction selection. The classic peephole-based instruction selector consists of a template-based expander that translates the compiler's IR into a more detailed form with a level of abstraction below the target ISA's level of abstraction; a simplifier that uses forward substitution, algebraic simplification, constant propagation, and dead-code elimination within a three or four operation scope; and a matcher that maps the optimized low-level IR onto the target ISA.

The strength of this approach lies in the simplifier; it removes interoperation inefficiencies that the expansion from compiler IR to low-level IR introduces. Those opportunities involve values that are local in scope; they cannot be seen at earlier stages of translation. The resulting improvements can be surprising. The final matching phase is straightforward; technologies ranging from hand-coded matchers to LR() parsers have been used.

Review Questions

1. Sketch a concrete algorithm for the simplifier that applies forward substitution, algebraic simplification, and local constant propagation. What is the complexity of your algorithm? How does the size of the peephole window affect the cost of running your algorithm over a block?
2. The example shown in Figure 11.10 on page 626 demonstrates one weakness of peephole-based selectors. The assignment of `2` to r_{10} is too far from the use of r_{10} to allow the simplifier to fold the constant and simplify the multiply (into either a `multI` or an `add`). What techniques might you use to expose this opportunity to the simplifier?

11.6 ADVANCED TOPICS

Both BURS-based and peephole-based instruction selectors have been designed for compile-time efficiency. Both techniques are limited, however, by the knowledge contained in the patterns that the compiler writer provides. To find the best instruction sequences, the compiler writer might consider using search techniques. The idea is simple. Combinations of instructions sometimes have surprising effects. Because the results are unexpected, they are rarely foreseen by a compiler writer and, therefore, are not included in the specification produced for a target machine.

Two distinct approaches that use exhaustive search to improve instruction selection have appeared in the literature. The first involves a peephole-based system that discovers and optimizes new patterns as it compiles code. The second involves a brute-force search of the space of possible instructions.

11.6.1 Learning Peephole Patterns

A major issue that arises in implementing or using a peephole optimizer is the tradeoff between the time spent specifying the target machine's instruction set and the speed and quality of the resulting optimizer or instruction selector. With a complete pattern set, the cost of both simplification and matching can be kept to a minimum by using an efficient pattern-matching technique. Of course, someone must generate all those patterns. On the other hand, systems that interpret the rules during simplification or matching have a larger overhead per LLIR operation. Such a system can operate with a much smaller set of rules. This makes the system easier to create. However, the resulting simplifier and matcher run more slowly.

One effective way to generate the explicit pattern table needed by a fast, pattern-matching peephole optimizer is to pair it with an optimizer that has

a symbolic simplifier. In this scheme, the symbolic simplifier records all the patterns it simplifies. Each time it simplifies a pair of operations, it records the initial pair and the simplified pair. Then, it can record the resulting pattern in the lookup table to produce a fast, pattern-matching optimizer.

The simplifier must check a proposed pattern against the machine description to ensure that the proposed simplification is broadly applicable.

By running the symbolic simplifier on a training set of applications, the optimizer can discover most of the patterns it needs. Then, the compiler can use the table as the basis of a fast pattern-matching optimizer. This lets the compiler writer expend computer time during design to speed up routine use of the compiler. It greatly reduces the complexity of the patterns that must be specified.

Increasing the interaction between the two optimizers can further improve code quality. At compile time, the fast pattern matcher will encounter some LLIR pairs that match no pattern in its table. When this occurs, it can invoke the symbolic simplifier to search for an improvement, bringing the power of search to bear only on the LLIR pairs for which it has no pre-existing pattern.

To make this approach practical, the symbolic simplifier should record both successes and failures. This allows it to reject previously seen LLIR pairs without the overhead of symbolic interpretation. When it succeeds in improving a pair, it should add the new pattern to the optimizer's pattern table, so that future instances of that pair will be handled by the more efficient mechanism.

This learning approach to generating patterns has several advantages. It applies effort only on previously unseen LLIR pairs. It compensates for holes in the training set's coverage of the target machine. It provides the thoroughness of the more expensive system while preserving most of the speed of the pattern-directed system.

In using this approach, however, the compiler writer must determine when the symbolic optimizer should update the pattern tables and how to accommodate those updates. Allowing an arbitrary compilation to rewrite the pattern table for all users seems unwise; synchronization and security issues are sure to arise. Instead, the compiler writer might opt for periodic updates—storing the newly found patterns away so they can be added to the table as a routine maintenance action.

11.6.2 Generating Instruction Sequences

The learning approach has an inherent bias: it assumes that the low-level patterns should guide the search for an equivalent instruction sequence. Some compilers have taken an exhaustive approach to the same basic problem.

Instead of trying to synthesize the desired instruction sequence from a low-level model, they adopt a generate-and-test approach.

The idea is simple. The compiler, or compiler writer, identifies a short sequence of assembly-language instructions that should be improved. The compiler then generates all assembly-language sequences of cost one, substituting the original arguments into the generated sequence. It tests each one to determine if it has the same effect as the target sequence. When it has exhausted all sequences of a given cost, it increments the cost of the sequences and continues. This process continues until (1) it finds an equivalent sequence, (2) it reaches the cost of the original target sequence, or (3) it reaches an externally imposed limit on either cost or compile time.

While this approach is inherently expensive, the mechanism used for testing equivalence has a strong impact on the time required to test each candidate sequence. A formal approach, using a low-level model of machine effects, is clearly needed to screen out subtle mismatches, but a faster test can catch the gross mismatches that occur most often. If the compiler simply generates and executes the candidate sequence, it can compare the results against those obtained from the target sequence. This simple approach, applied to a few well-chosen inputs, should eliminate most of the inapplicable candidate sequences with a low-cost test.

This approach is, obviously, too expensive to use routinely or to use for large code fragments. In some circumstances, however, it merits consideration. If the application writer or the compiler can identify a small, performance-critical section of code, the gains from an outstanding code sequence may justify the cost of exhaustive search. For example, in some embedded applications, the performance-critical code consists of a single inner loop. Using exhaustive search for small code fragments—to improve either speed or space—may be worthwhile.

Similarly, exhaustive search has been applied as part of the process of retargeting a compiler to a new architecture. This application uses exhaustive search to discover particularly efficient implementations for IR sequences that the compiler routinely generates. Since the cost is incurred when the compiler is ported, the compiler writer can justify the use of search by amortizing that cost over the many compilations that are expected to use the new compiler.

11.7 SUMMARY AND PERSPECTIVE

At its heart, instruction selection is a pattern-matching problem. The difficulty of instruction selection depends on the level of abstraction of the compiler's IR, the complexity of the target machine, and the quality of code

desired from the compiler. In some cases, a simple treewalk approach will produce adequate results. For harder instances of the problem, however, the systematic search conducted by either tree-pattern matching or peephole optimization can yield better results. Creating a handcrafted treewalk code generator that achieves the same results would take much more work. While these two approaches differ in almost all their details, they share a common vision—the use of pattern matching to find a good code sequence among the myriad sequences possible for any given IR program.

Tree-pattern matchers discover low-cost tilings by taking the low-cost choice at each decision point. The resulting code implements the computation specified by the IR program. Peephole transformers systematically simplify the IR program and match what remains against a set of patterns for the target machine. Because they lack explicit cost models, no argument can be made for their optimality. They generate code for a computation with the same effects as the IR program, rather than a literal implementation of the IR program. Because of this subtle distinction in the two approaches, we cannot directly compare the claims for their quality. In practice, excellent results have been obtained with each approach.

The practical benefits of these techniques have been demonstrated in real compilers. Both LCC and GCC run on many platforms. The former uses tree-pattern matching; the latter uses a peephole transformer. The use of automated tools in both systems has made them easy to understand, easy to retarget, and, ultimately, widely accepted in the community.

Equally important, the reader should recognize that both families of automatic pattern matchers can be applied to other problems in compilation. Peephole optimization originated as a technique for improving the final code produced by a compiler. In a similar way, the compiler can apply tree-pattern matching to recognize and rewrite computations in an AST. BURS technology can provide a particularly efficient way to recognize and improve simple patterns, including the algebraic identities recognized by value numbering.

■ CHAPTER NOTES

Most early compilers used hand-coded, ad hoc techniques to perform instruction selection [26]. With sufficiently small instruction sets, or large enough compiler teams, this worked. For example, the BLISS-11 compiler generated excellent code for the PDP-11, with its limited repertoire of operations [356]. The small instruction sets of early computers and minicomputers let researchers and compiler writers ignore some of the problems that arise on modern machines.

For example, Sethi and Ullman [311], and, later, Aho and Johnson [5] considered the problem of generating optimal code for expression trees. Aho, Johnson, and Ullman extended their ideas to expression DAGs [6]. Compilers based on this work used ad hoc methods for the control structures and clever algorithms for expression trees.

In the late 1970s, two distinct trends in architecture brought the problem of instruction selection to the forefront of compiler research. The move from 16- to 32-bit architectures precipitated an explosion in the number of operations and address modes that the compiler had to consider. For a compiler to explore even a large fraction of the possibilities, it needed a more formal and powerful approach. At the same time, the nascent Unix operating system began to appear on multiple platforms. This sparked a natural demand for C compilers and increased interest in retargetable compilers [206]. The ability to easily retarget the instruction selector plays a key role in determining the ease of porting a compiler to new architectures. These two trends started a flurry of research on instruction selection that started in the 1970s and continued well into the 1990s [71, 72, 132, 160, 166, 287, 288].

The success of automation in scanning and parsing made specification-driven instruction selection an attractive idea. Glanville and Graham mapped the pattern matching of instruction selection onto table-driven parsing [160, 165, 167]. Ganapathi and Fischer attacked the problem with attribute grammars [156].

Tree-pattern-matching code generators grew out of early work in table-driven code generation [9, 42, 167, 184, 240] and in tree-pattern matching [76, 192]. Pelegri Llopart formalized many of these notions in the theory of BURS [281]. Subsequent authors built on this work to create a variety of implementations, variations, and table-generation algorithms [152, 153, 288]. The Twig system combined tree-pattern matching and dynamic programming [2, 334].

The first peephole optimizer appears to be McKeeman's system [260]. Bagwell [30], Wulf et al. [356], and Lamb [237] describe early peephole systems. The cycle of expand, simplify, and match described in Section 11.5.1 comes from Davidson's work [115, 118]. Kessler also worked on deriving peephole optimizers directly from low-level descriptions of target architectures [222]. Fraser and Wendt adapted peephole optimization to perform code generation [154, 155]. The machine learning approach described in Section 11.6.1 was described by Davidson and Fraser [116].

Massalin proposed the exhaustive approach described in Section 11.6.2 [258]. It was applied in a limited way in GCC by Granlund and Kenner [170].

EXERCISES

1. The treewalk code generator shown in Figure 7.2 uses a `loadI` for every number. Rewrite the treewalk code generator so that it uses `addI`, `subI`, `rsubI`, `multI`, `divI` and `rdivI`. Explain any additional routines or data structures that your code generator needs. Section 11.2

2. Using the rules given in Figure 11.5, generate two tilings for the AST shown in Figure 11.4. Section 11.3

3. Build a low-level AST for the following expressions, using the tree in Figure 11.4 as a model:
a. `y ← a × b + c × d`
b. `w ← a × b × c - 7`
Use the rules given in Figure 11.5 to tile these trees and generate ILOC.

4. Tree-pattern matching assumes that its input is a tree.
a. How would you extend these ideas to handle DAGs, where a node can have multiple parents?
b. How do control-flow operations fit into this paradigm?

5. In any treewalk scheme for code generation, the compiler must choose an evaluation order for the subtrees. That is, at some binary node *n*, does it evaluate the left subtree first or the right subtree first?
a. Does the choice of order affect the number of registers required to evaluate the entire subtree?
b. How can this choice be incorporated into the bottom-up tree-pattern matching schemes?

6. A real peephole optimizer must deal with control-flow operations, including conditional branches, jumps, and labelled statements. Section 11.4
a. What should a peephole optimizer do when it brings a conditional branch into the optimization window?
b. Is the situation different when it encounters a jump?
c. What happens with a labelled operation?
d. What can the optimizer do to improve this situation?

7. Write down concrete algorithms for performing the simplification and matching functions of a peephole transformer.
a. What is the asymptotic complexity of each of your algorithms?
b. How is the running time of the transformer affected by a longer input program, by a larger window, and by a larger pattern set (both for simplification and for matching)?

8. Peephole transformers simplify the code as they select a concrete implementation for it. Assume that the peephole transformer runs

before either instruction scheduling or register allocation and that the transformer can use an unlimited set of virtual register names.

a. Can the peephole transformer change the demand for registers?

b. Can the peephole transformer change the set of opportunities that are available to the scheduler for reordering the code?