Chapter 11

Instruction Selection

CHAPTER OVERVIEW

The compiler’s front end and optimizer both operate on the code in its IR form. To create executable code, the compiler must rewrite it into the processor’s instruction set. This process is called instruction selection. Because a typical processor provides multiple ways to express most computations, the selector must choose the best sequence from multiple implementations.

This chapter introduces two different approaches to instruction selection. The first uses 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 an 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 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, and tools that implement them.
**Conceptual Roadmap**

Instruction selection, which maps code in the compiler’s IR into the target ISA, is a pattern-matching problem. At its simplest, the compiler could provide a single assembly code sequence for each IR operation. The resulting selector would produce correct code using a template-like expansion. Unfortunately, that code would likely make poor use of the target machine’s resources. Better approaches consider multiple code sequences for each IR operation, along with the operation’s context, to 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 to describe both the compiler’s IR and the target machine’s ISA. The latter approach translates the compiler’s IR into a low-level linear IR, systematically improves that IR, and then maps the improved IR into the target machine’s ISA. Each of these techniques can produce high-quality code that is tailored to local context. Each has been incorporated into tools that take a target machine description and produce a working instruction selector.

**A Few Words About Time**

The instruction selector, itself, runs at compile time to translate the IR program created by the front end and optimizer into code expressed in the target machine’s ISA. The code produced by selection may not be a valid assembly program; in many compilers, the post-selection code still assumes an unlimited supply of virtual registers. The register allocator, which runs after the instruction selector at compile time, rewrites the code into the target ISA’s register set.

Many compilers use description-based instruction selectors, based on either tree-pattern matching (see Section 11.4) or peephole optimization (see Section 11.3). With these tools the compiler writer creates, at design time, descriptions of the compiler’s IR and the target ISA. At build time, a back-end generator analyzes these descriptions and produces code for the instruction selector, which is then compiled and included in the compiler.

**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 components of the compiler.
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 problem. 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 complex register use rules, then the selector may need to 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.

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 the source languages that it supports. It optimizes the IR 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; a description of each operation; 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. The selector consumes the compiler’s IR and produces assembly code for the target machine. A retargetable instruction selector consists of a pattern-matching engine coupled to information about the IR, the ISA, and the mapping between them.

If we can automate the construction of the matching engine, construction of instruction selectors can follow the same model as specification-driven scanners and parsers. The compiler writer creates a description of the target machine and a back-end generator constructs the actual instruction selector. As with scanners and parsers, the generator runs offline during compiler development.

This approach moves the cost and complexity of instruction selection into the back-end generator. Just as in LR parsing, we can afford to use algorithms in the generator that require more time than the algorithms that run at compile time. In fact, tool-based instruction selectors can be extremely efficient at compile time.

A second key to retargetability is to isolate machine-dependent code to the greatest extent possible. Ideally, all machine-dependent code should appear in the instruction selector, scheduler, and register allocator; unfortunately, the reality almost always falls 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.

Inevitably, the compiler writer must decide how much customization of the code should occur in the compiler’s back end. We assume that the optimizer has already improved any opportunities that it can discover. In this view, the focus of instruction selection is to provide a good local mapping from the details of the IR operations into the target machine’s instruction set. Logic and experience suggest that we separate the goal of optimizing computation from the goal of mapping that computation efficiently onto the target machine.

This chapter examines two approaches to automating the construction of instruction selectors. Section 11.2 explores background issues in instruction selection at greater depth and introduces the example that we will use throughout the chapter. The two subsequent sections present different ways to apply pattern-matching techniques to transform IR sequences to assembly sequences. The first technique, in Section 11.3, builds on ideas from peephole optimization. The sec-
ond technique, in Section 11.4, builds on algorithms for matching tree patterns against trees. Both of these methods are description based. The compiler writer creates 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.

## 11.2 Background

Instruction selection discovers a way to express the IR computation in the target machine’s ISA. If the IR and ISA are both written at a similar level of abstraction, then the translation from IR to ISA may be straightforward, as in translating an ILOC program to run on a simple RISC machine. If, on the other hand, the IR is more abstract than the ISA, as with a source-level AST and a commodity microprocessor, then the selector needs to supply additional detail. If the IR is less abstract than the ISA, as with the register-transfer language used in early versions of GCC, then the selector may need to combine multiple IR operations into a single target machine operation.

The compiler writer’s goals come into play, as well. The compiler writer must confront the tradeoff between complexity in the compiler’s back end and the performance of the compiled code. It takes a more complex compiler to produce customized target-machine code than to produce slower template-like code. Each approach might make sense in an appropriate context.

Chapter 5 discussed the issues that arise when the compiler translates source code into an initial IR. Instruction selection differs from IR generation in two major ways. First, IR generation should produce a version of the program that compiles and optimizes well, while instruction selection should produce a version that runs well. Second, the compiler writer has control over the design of the IR, which can simplify the task of mapping source code into IR; the target ISA is a fixed language bound by limited resources. Because of these differences, code generation in a compiler’s back end uses more complex approaches and algorithms than does IR generation in its front end.

The complexity of instruction selection arises not from a particular methodology or a specific matching algorithm, but rather, from the nature of the underlying problem—a processor typically provides multiple ways to implement an IR construct, each with its own costs and its own restrictions. The code generator must choose among these alternatives based on knowledge of operation costs and surrounding context. Done well, this process should produce efficient code that is customized to fit well with the surrounding operations.

Specification-based tools can move most of the complexity of generating customized instruction sequences from compile time back
Runs Well Versus Optimizes Well

The difference between generating IR early in the compiler and generating assembly code late in the compiler lies in the intended use of the code. The IR form of the program is intended for optimization and code generation. The assembly code version is intended for execution. To see how these different use cases drive the compiler to different code sequences, consider the following abstracted loop.

```plaintext
for i = 1 to 1000
  for j = 1 to 1000
    ...A[i,j] ...
```

Section ?? presented two ways to compute the address of \( A[i,j] \), assuming that \( A \) is dimensioned as \( A[low_1:high_1, low_2:high_2] \) and stored in row-major order:

1. \( @A + (i - low_1) \times len_2 \times w + (j - low_2) \times w \)
2. \( @A + (i \times len_2 \times w) + (j \times w) - (low_1 \times len_2 \times w + low_2 \times w) \).

The former polynomial uses fewer operations; it will execute more quickly than the latter one. The latter polynomial separates out subexpressions that depend on \( i \) and \( j \), plus a term that contains only constants. A good optimizer might move the \( i \) term out of the inner loop and evaluate the constant term at compile time. Thus, formula (1) runs well while formula (2) optimizes well.

into design time and build time. Such tools automatically construct efficient and effective instruction selectors that manage the complexity of context in both the IR and the ISA. They can simplify construction of an effective compiler back end, in much the same way that scanner generators and parser generators simplify front-end construction.

11.2.1 The Impact of ISA Design on Selection

Much of the complexity of instruction selection arises directly from properties of a typical target machine ISA. Two factors that cause an explosion in the number of cases that the selector must consider are: duplicate mechanisms to accomplish a single task and the proliferation of address modes in memory and arithmetic operations.

Features such as these complicate instruction selection, because they add context, they add complexity and they add choices. Automated techniques for building instruction selection arose, in large part, as a response to increasing processor complexity. By specify-
ing the IR to ISA mapping in a more concise way and algorithmically expanding that specification into code, the tools simplify the task of building efficient and powerful selectors.

**Duplicate Implementations**

If the target ISA provided just one way to perform each IR operation, 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, a simple copy operation $r_i \rightarrow r_j$. Assume that the target processor uses ILOC as its native instruction set. ILOC is simple, but even it exposes the complexity 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,

\begin{align*}
\text{addI } r_i,0 & \Rightarrow r_j \\
\text{subI } r_i,0 & \Rightarrow r_j \\
\text{multI } r_i,1 & \Rightarrow r_j \\
\text{divI } r_i,1 & \Rightarrow r_j \\
\text{lshiftI } r_i,0 & \Rightarrow r_j \\
\text{rshiftI } r_i,0 & \Rightarrow r_j \\
\text{and } r_i, r_i & \Rightarrow r_j \\
\text{orI } r_i,0 & \Rightarrow r_j \\
\text{xorI } r_i,0 & \Rightarrow r_j
\end{align*}

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. If we consider two-operation sequences, the set is even larger.

A programmer would discount most, if not all, of these alternative sequences. A register-to-register copy operation, such as $i2i$, is simple, fast, and obvious. An automated process, however, may need to consider all the possibilities and make the best choices. The fact that an ISA can accomplish an effect in multiple ways makes instruction selection harder. Even ILOC, a particularly simple instruction set, allows many ways to implement a copy operation.

**Address Modes**

The arithmetic operations shown above all assumed a register-to-register model. Some processors provide variants of the arithmetic operators that specify one or more operands with a memory address. Other operators, such as load and store, must address memory. To improve the efficiency, ISAs introduce *address modes* that encode common address computations.

For example, the ARM V8 has a zero register, XZR.
ILOC models this situation by providing multiple address modes on load operations:

<table>
<thead>
<tr>
<th>Operation</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>load $r_1 \Rightarrow r_2$</td>
<td>MEMORY($r_1$) $\Rightarrow$ $r_2$</td>
</tr>
<tr>
<td>loadAI $r_1, c_2 \Rightarrow r_3$</td>
<td>MEMORY($r_1 + c_2$) $\Rightarrow$ $r_3$</td>
</tr>
<tr>
<td>loadAO $r_1, r_2 \Rightarrow r_3$</td>
<td>MEMORY($r_1 + r_2$) $\Rightarrow$ $r_3$</td>
</tr>
<tr>
<td>loadI $c_1 \Rightarrow r_2$</td>
<td>$c_1 \Rightarrow r_2$</td>
</tr>
</tbody>
</table>

Store operations support the address-immediate (AI) and address-offset (AO) modes. Arithmetic operations also support the immediate (I) mode so that they can read one operand as a literal constant embedded in the instruction stream.

Real processors often support a larger and more complex set of address modes. Examples include arithmetic operations that specify one or more arguments from memory; one or two address operations that overwrite some of their input arguments; and operations with implicit arguments, such as the top of a hardware-supported runtime stack. The instruction selector should make effective use of any and all address modes supported by the target ISA.

Similar issues arise in control-flow operations. Branches and jumps may support absolute addresses, PC-relative addresses, and addresses of different bit-lengths. These operations may be of different sizes in memory; they may take different numbers of cycles. Because the selection of the best form of branch or jump depends on multiple factors, including the distance from source to destination, selecting branch addressing modes requires painstaking care.

**Level of Abstraction** The individual operations in ILOC have a relatively low level of abstraction. Modern processors often provide a mix of low-level operations and more complex operations. For example:

- A string-move operation allows the code to easily specify a complex sequence that includes an implicit iterative loop.
- A procedure-call operation might automate large parts of the call sequence, including management of caller saves registers.
- A floating point multiply-add operation might use fewer cycles and fewer registers to compute ($r_i \times r_j$) + $r_k$ than the individual multiply and add operations.
- A load-multiple or store-multiple operation might move values into or out of several contiguous registers.

Operations such as these require the instruction selector to synthesize several low-level operations into one higher-level operation.
Register Use  Processors place idiosyncratic constraints on register use (see Section 13.2.4). Operations may expect one or more operands in designated locations. On some processors, double-precision values must be stored in a pair of floating point registers that begin with an even-numbered register. The use of a destructive one-address or two-address operation can kill the value of one of its arguments, which complicates their use (see Section 4.3).

Costs  Each operation has its own cost. Most modern machines implement simple operations, such as an addition or a shift as single-cycle operations. Other operations, such as a load or a divide, may take longer. The latency of multiplication and division may depend on the bit patterns in the operands. The latency of a memory operation depends on many factors, including the detailed current state of the computer’s memory system. The latency of a branch may depend on how well the processor can predict its outcome.

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 consume different amounts of energy. A compiler that tries to optimize for energy may use radically different costs than would one optimizing for speed. Similarly, if code space is critical, the compiler writer might assign costs based solely on sequence length. Alternatively, the compiler writer might simply exclude from consideration all multi-operation sequences that achieve the same effect as a single-operation sequence.

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. Even moderately-sized instruction sets can produce search spaces that contain 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.

11.2.2 Motivating Example

The discussions of peephole optimization and tree-pattern matching as technologies for instruction selection use the same example to motivate, explain, and explore the issues. In both cases, the sections examine how to generate code for the simple assignment statement: \( a ← b - 2 \times c \). Figure 11.1 shows the example in two different IRs. Panel (a) shows the same statement in quadruple form; the discussion

For example, on IA-32, a 16-bit signed integer multiply reads one argument from \( AX \) and writes its 32-bit result into \( AX \) and \( DX \).

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

This same example appears in Figure 4.1 on page 157.
of peephole-based instruction selection starts from this IR. Panel (b) shows a low-level AST for the statement; the discussion of tree-pattern matching uses this IR.

The example uses three variables, each with a different kind of address computation. $a$ is a local variable that resides at offset 4 from the ARP. $b$ is a call-by-reference parameter that resides at offset $-16$ from the ARP. $c$ is a global variable.

The address computation for $c$ is complex. The value resides at offset 12 from a global label. Because the label resolves to a full-length address, the resolved address is stored in some global constant pool. The constant pool starts at $\oplus CP$ and the offset in the pool is given by $\oplus G$. We assume that $\oplus CP$ is too large for an immediate field in an add or sub, but small enough for the immediate field in a loadI.

Because the example consists of a single source-level statement, it contains no control flow. Thus, we will assume that any condition-code values defined by the operations are unused and therefore dead.

**Quadruples** Figure 11.1.a shows the example code expressed in classic quadruple form. Variable names appear as references, with additional detail available in the symbol table. Operations are simple three-address code. A compiler-generated temporary name is used to carry the result of the multiply into the addition. The order of the quadruples encodes the execution order.

This representation has little explicit detail. It assumes a symbol table that contains necessary detail on all the named values (see Sections 4.4 and 5.4). As we will see in Section 11.3, the peephole instruction selector will immediately expand this code to ensure that the needed detail is explicit and exposed.
**Low-Level AST**  To expose enough detail for instruction selection, the AST shown in Figure 11.1.b has a low-level of abstraction. Several of the nodes in the tree need further explanation.

Constant values are represented by three distinct kinds of nodes:

- A NUM node represents a constant that fits into the immediate field of a three-operand immediate instruction (e.g., `mul t1`).
- A CON node, not shown in this example, represents a constant that is small enough to fit in a `loadI`, but too large for a NUM.
- A LAB node represents a relocatable symbol, typically an assembly-level label used for either code or data.

The distinction between these kinds of constants is critical to instruction selection. A value in a CON or LAB node cannot appear as an immediate operand in a `mul t1` operation. A value represented by a LAB node is assumed to be too large to fit in a `loadI` operation; the compiler groups such nodes into a constant pool in memory.

Two other nodes in this tree have non-obvious meanings. A VAL node represents a value known to reside in a register, such as the ARP in `r_{arp}`, or the result of evaluating a common subexpression identified by the optimizer. A ◆ node signifies a level of indirection; its child is an address and it produces the value stored at that address.

The low-level detail in the AST allows the instruction selector to tailor its decisions to specific context. The subtrees that describe the address expressions for `a`, `b`, and `c` all look similar; as we shall see, they generate distinctly different code due to the specific base addresses and offsets. By tailoring the final assembly code to context, the compiler can produce efficient code for each subtree.

### 11.2.3 Ad Hoc Matching

The compiler writer can construct an instruction selector around a hand-written, ad hoc matcher. Such a selector would recognize individual IR constructs or small sets of related constructs and map them directly into the assembly language (ASM) of the target machine. For a tree-based IR, this approach might result in a simple postorder walk, similar to the order in which the syntax-driven translation schemes of Chapter 5 generated IR from the parser. For a linear IR, the compiler writer might produce a selector that makes a linear scan over the IR and emits code for each operation.

Instruction selectors built around ad hoc matching tend to produce more uniform, template-like code. To take deeper advantage of the target machine’s ISA requires the aggregation of context from the IR program. The selector can only make effective use of complex address modes if it builds up some representation, implicit or explicit,
of the address computation, and uses that model when it selects the form of a load or store operation. The selector can only make use of immediate-mode operations if it understands, even in limited ways, the flow of constant values through the code and the magnitudes of those values.

Finally, ad hoc matching provides little, if any, support for retargeting the compiler to another processor. Portability is a key rationale for isolating the instruction selector both from optimization and from register allocation and instruction scheduling. A tool-based approach to specification-driven instruction selectors should simplify the task of retargeting the compiler to a new processor or system.

SECTION REVIEW
A compiler’s instruction selector rewrites code expressed in the compiler’s IR into code expressed in the target machine’s assembly code. The selector must, for each IR construct, choose an efficient assembly-language implementation, tailored to the local context surrounding the construct. It may need to change the level of abstraction of the code. The selector itself must run quickly at compile time.

Instruction selection is hard because of the sheer number of choices that the selector confronts. It must choose from multiple operations. It must make effective use of the target’s address modes. It must model any restrictions on processor resource use and ensure that the generated code respects them. Because each choice affects the runtime efficiency of the compiled code, the selector plays a critical role in application performance.

REVIEW QUESTIONS
1. Some ISAs include operations that use memory-based operands. Consider a memory-to-memory move operation \( \text{move } r_i \Rightarrow r_k \), where \( r_i \) and \( r_k \) are addresses. The move operation copies four bytes from memory address \( r_i \) to memory address \( r_k \). It costs one more cycle than the equivalent load operation.
   a. What advantages might such an operation present?
   b. Where in the compiler might the decision to use such an operation best be made?

2. Enumerate as many ways as you can in ILOC to set register \( r_1 \) to contain the integer value one.
11.4 SELECTION VIA TREE-PATTERN MATCHING

Another way to attack the complexity of instruction selection is with the use of tree-pattern matching tools. To transform code generation into a tree-pattern matching problem, both the IR form of the program and the target machine’s instruction set must be expressed as trees. 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 modeled by operation trees like those shown in the left margin for add and addI. By systematically matching such operation trees, or pattern trees, with subtrees of an AST, the compiler can discover all the potential implementations for the subtree.

Given a low-level AST for the code and a collection of operation trees for the target machine’s ISA, the matcher constructs a tiling of the AST with operation trees. A tile is a pair, \( \langle x, y \rangle \), where \( x \) is a node in the AST and \( y \) is the root of an operation tree. A tiling is a set of pairs that meet the following constraints:

1. The set of tiles covers every AST node.
2. The root of each operation tree overlays a leaf in another operation tree, unless it overlays the root of the AST.
3. Where two operation trees overlap, they are compatible; that is, they agree in both storage class and value type.
4. The overlap between any two operation trees in the set occurs at a single node.

A set that meets these criteria is a tiling; it implements the AST.

The presence of a pair \( \langle x, y \rangle \) in the tiling indicates that the AST subtree rooted at \( x \) can be implemented by the operation tree rooted at \( y \). Unless \( x \) is a leaf, the choice of \( y \) will depend on finding implementations for subtrees of \( x \) that work with \( y \). To build a tiling, the compiler must ensure that both the entire tree and each of its subtrees can be implemented by the specified set of operation trees.

Given a tiling that implements an AST, the compiler can generate assembly code in a bottom-up walk over the AST. Thus, the key to making tree-pattern matching 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. These algorithms associate costs with the operation trees and produce minimal cost tilings. They differ in the specific 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 vary during matching. This section focuses on tree matching, but the insights carry over to text matching and bottom-up rewrite systems.
11.4.1 Representing Trees

We need a notation to describe both ASTs and tree patterns. Prefix notation is ideal for this task. For example, the add operation shown in the margin is \((\text{Reg}_i, \text{Reg}_j)\) in prefix form, and the addI operation is \((\text{CON}_i, \text{Reg}_j)\). This same notation works for both low-level ASTs that represent executable code and the pattern trees that represent target-machine operations. The examples in this chapter limit themselves to integer operations. Extending the rules to other data types adds many new patterns, but few new insights.

The operands of a subtree are either subtrees or leaves. Subtrees are expressed in prefix notation, as with the multiply subtree in the AST shown in the margin: \((\times (\text{CON}_2, \text{Reg}_y), \text{Reg}_x)\). Leaves are assigned symbolic names that encode information about the type and storage location of the operand. For example, \text{Reg}_i\ indicates a value that resides in a register and \text{CON}_j\ indicates a generic constant value. In an AST, labels such as \text{VAL}, \text{NUM}, \text{LAB}, and \text{CON}\ provide more detailed information (see page 597). Subscripts are added to names for uniqueness. If we rewrite the AST from Figure 11.1.a in prefix form, it becomes:

\[
\leftarrow (\text{VAL}_1, \text{NUM}_1), \\
- (\text{LAB}_1, \text{LAB}_2, \text{LAB}_3, \text{LAB}_4)
\]

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

Throughout the discussion, we will work with virtual names; that is, we assume that the compiler can use as many names as it needs. After selection, a register allocator will map these “virtual” names onto the target machine’s limited set of physical registers (see Chapter 13).

11.4.2 Rewrite Rules

To build an instruction selector based on tree-pattern matching, the compiler writer creates a set of rewrite rules that maps the AST into the target machine’s ISA. Each rule consists of a tree pattern, a code template, and an associated cost. These tree patterns describe the structure of the AST. Figure 11.4 shows the ILOC subset that we will use in the ongoing examples.

The patterns resemble productions in a context-free grammar (CFG). Each pattern has a left-hand side (LHS) and a right-hand side (RHS); as in a context-free grammar, the LHS and RHS are separated by the derives symbol, \(\rightarrow\). The drawing in the margin shows the pattern for three register add, along with the tree that it represents. The labels give names to the various definitions and uses. The input arguments are \text{Reg}_1\ and \text{Reg}_2, respectively. The output of the operation is \text{Reg}_0.
### Arithmetic Operations

<table>
<thead>
<tr>
<th>Operation</th>
<th>Code</th>
<th>Memory Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>( r_1, r_2 \Rightarrow r_3 )</td>
<td>store ( r_1 \Rightarrow r_2 )</td>
</tr>
<tr>
<td>addI</td>
<td>( r_1, c_2 \Rightarrow r_3 )</td>
<td>storeAO ( r_1 \Rightarrow r_2,r_3 )</td>
</tr>
<tr>
<td>sub</td>
<td>( r_1, r_2 \Rightarrow r_3 )</td>
<td>storeAI ( r_1 \Rightarrow r_2,c_3 )</td>
</tr>
<tr>
<td>subI</td>
<td>( r_1, c_2 \Rightarrow r_3 )</td>
<td>loadI ( c_1 \Rightarrow r_3 )</td>
</tr>
<tr>
<td>rsubI</td>
<td>( r_2,c_1 \Rightarrow r_3 )</td>
<td>load ( r_1 \Rightarrow r_3 )</td>
</tr>
<tr>
<td>mult</td>
<td>( r_1, r_2 \Rightarrow r_3 )</td>
<td>loadAO ( r_1,r_2 \Rightarrow r_3 )</td>
</tr>
<tr>
<td>multI</td>
<td>( r_1, c_2 \Rightarrow r_3 )</td>
<td>loadAI ( r_1,c_2 \Rightarrow r_3 )</td>
</tr>
</tbody>
</table>

### Figure 11.4 The ILOC Subset

Some symbols appear exclusively on the RHS of patterns. These symbols are analogous to the terminal symbols in a CFG; they represent concrete symbols that can be leaves in a tree. Other symbols appear on either the LHS or RHS of a pattern. These symbols are analogous to the non-terminal symbols in a CFG; they are syntactic variables used to tie productions together into sequences.

A rule set consists of a collection of related rules that, together, can tile a tree. Figure 11.5 shows a rule set for the low-level AST that we described in Section 11.2.2. Each rule represents a small AST. For example, the rule shown earlier in the margin is rule 8. Its RHS consists of the \( + \) node and its two children, while the label on the \( + \) node, \( \text{Reg} \), represents the rule’s LHS.

Rule 19 deals with a common problem: the code needs to load a constant value that is too large to accommodate in a load immediate operation. The code template suggests one way to solve the problem. It assumes that each procedure has a unique, statically initialized constant pool, and it represents the start of that constant pool with the symbol \( @\text{CP} \). (In practice, \( @\text{CP} \) would undoubtedly be constructed by mangling the name of the procedure.) Further, it assumes that \( @\text{L} \) is the positive offset of \( \text{LAB}_1 \) from \( @\text{CP} \). (In practice, the instruction selector would need to lookup \( \text{LAB}_1 \) in a table to find the value of \( @\text{L} \).) With these assumptions, the emitted code loads \( @\text{CP} \) into a register and uses it as the base address for a loadAI operation.

The rule set in Figure 11.5 describes the set of potential ASTs for a list of assignment statements. Not all the rules describe code-producing trees; for example, rules 0 and 1 create a sequence of \( \text{Stmt} \), the non-terminal symbol for an assignment statement; rules 20 through 23 handle address mode selection, and rule 24 is used in immediate-mode operations.

Interactions between the patterns, encoded through the use of LHS symbols, define the ways in which subtrees can combine. For exam-
11.4 Selection via Tree-Pattern Matching

<table>
<thead>
<tr>
<th>Production</th>
<th>Cost</th>
<th>Code Template</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 Goal → Goal Stmt</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1 Goal → Stmt</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2 Stmt → ← (Reg₁,Reg₂)</td>
<td>3</td>
<td>store r₂ ⇒ r₁</td>
</tr>
<tr>
<td>3 Stmt → ← (T₁₁,Reg₂)</td>
<td>3</td>
<td>storeAO r₂ ⇒ T₁.r₁,T₁.r₂</td>
</tr>
<tr>
<td>4 Stmt → ← (T₂₁,Reg₂)</td>
<td>3</td>
<td>storeAI r₂ ⇒ T₂.r₁,T₂.n</td>
</tr>
<tr>
<td>5 Reg → ◆ (Reg₁)</td>
<td>3</td>
<td>load r₁ ⇒ rₙₑw</td>
</tr>
<tr>
<td>6 Reg → ◆ (T₁)</td>
<td>3</td>
<td>loadAO T₁.r₁,T₁.r₂ ⇒ rₙₑw</td>
</tr>
<tr>
<td>7 Reg → ◆ (T₂₁)</td>
<td>3</td>
<td>loadAI T₂.r₁,T₂.n ⇒ rₙₑw</td>
</tr>
<tr>
<td>8 Reg → +(Reg₁,Reg₂)</td>
<td>1</td>
<td>add r₁,r₂ ⇒ rₙₑw</td>
</tr>
<tr>
<td>9 Reg → +(Reg₁,T₃₂)</td>
<td>1</td>
<td>addI r₁,T₃ ⇒ rₙₑw</td>
</tr>
<tr>
<td>10 Reg → +(T₃₁,Reg₂)</td>
<td>1</td>
<td>addI r₂,T₃ ⇒ rₙₑw</td>
</tr>
<tr>
<td>11 Reg → -(Reg₁,Reg₂)</td>
<td>1</td>
<td>sub r₁,r₂ ⇒ rₙₑw</td>
</tr>
<tr>
<td>12 Reg → -(Reg₁,T₃₂)</td>
<td>1</td>
<td>subI r₁,T₃ ⇒ rₙₑw</td>
</tr>
<tr>
<td>13 Reg → -(T₃₁,Reg₂)</td>
<td>1</td>
<td>rsubI r₂,T₃ ⇒ rₙₑw</td>
</tr>
<tr>
<td>14 Reg → × (Reg₁,Reg₂)</td>
<td>2</td>
<td>mult r₁,r₂ ⇒ rₙₑw</td>
</tr>
<tr>
<td>15 Reg → × (Reg₁,T₃₂)</td>
<td>2</td>
<td>multI r₁,T₃ ⇒ rₙₑw</td>
</tr>
<tr>
<td>16 Reg → × (T₃₁,Reg₂)</td>
<td>2</td>
<td>multI r₂,T₃ ⇒ rₙₑw</td>
</tr>
<tr>
<td>17 Reg → CON₁</td>
<td>1</td>
<td>loadI CON₁ ⇒ rₙₑw</td>
</tr>
<tr>
<td>18 Reg → NUM₁</td>
<td>1</td>
<td>loadI NUM₁ ⇒ rₙₑw</td>
</tr>
<tr>
<td>19 Reg → LAB₁</td>
<td>4</td>
<td>loadI @CP ⇒ rₙₑw; loadI rₙₑw, φL ⇒ rₙₑw</td>
</tr>
<tr>
<td>20 Reg → VAL₁</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>21 T₁ → +(Reg₁,Reg₂)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>22 T₂ → +(Reg₁,T₃₂)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>23 T₂ → +(T₃₁,Reg₂)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>24 T₃ → NUM₁</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 11.5** Rewrite Rules for Tiling the Low-Level Tree with ILOC

...
to name the value. The VAL might be a global value, such as the ARP, or it might be the result of a computation performed in a disjoint subtree, such as a redundant expression found by the optimizer.

T1 and T2 represent addresses—values that can be computed in address modes such as loadA0 and loadAI. T3 represents a NUM used directly as an immediate operand.

Finally, note that the rule set is ambiguous. Rules 9 and 22 have the same RHS, as do rules 10 and 23. Those pairs have different purposes and different costs. Rule 9 represents an explicit, general purpose add operation, while rule 22 represents an add operation done by the memory addressing hardware.

Cost Estimates Each rule has a cost; that cost should provide a realistic estimate of the runtime cost of executing the code in the template. Figure 11.5 assumes that addition and subtraction require one cycle, multiplication takes two, and memory operations need three cycles, a reasonable estimate for a hit in the first-level data cache. Rules that generate no code, such as rule 22, have zero cost. The code generator will use these costs to choose among the possible alternatives.

Notice that rule 19 emits multiple operations. It matches a LAB, which must be retrieved from memory at some constant offset from the current procedure’s constant pool. The constant pool resides at @CP; the offset from @CP is given by @L.

Building a Rule Set A common strategy in constructing a rule set is to begin with a minimal set of rules that covers the tree. That is, the compiler writer can derive a set of rules that has a pattern for every node in the AST and a correct code sequence for each pattern. Once that rule set has been designed, tested, and debugged, the compiler writer can add rules with patterns that use specialized operations to handle more complex cases.

Restrictions on the Rule Set We restricted the rules in Figure 11.5 in two specific ways. First, each pattern includes at most one operator. This restriction significantly simplifies the code for the tree-pattern matcher in Section 11.4.3 without changing its expressiveness.

To see this, consider two cases shown in Figure 11.6. Panel (a) shows a single pattern for \( \leftarrow (+ (\text{Reg}_1, \text{Reg}_2)) , \text{Reg}_3 \). This pattern specializes the general assignment described by rule 2 for the case when the address is the sum of two registers. Panel (b) shows two single-operator patterns that together tile the same subtree. Any multi-operator pattern can be rewritten as a series of single-operator patterns, at the cost of introducing new LHS symbols.

This single-operator restriction significantly simplifies the code for the tree-pattern matcher. To match the pattern in panel (a), the
matcher must inspect both the + node and the ← node together, or it must carry along explicit state to track such matches. Either approach adds complexity and cost to the implementation.

With the two single-operator patterns in panel (b), the matcher makes purely local decisions. The AST subtree + (Reg,Reg) matches both the general rule that describes an add (rule 8) and the rule that describes a T1 (rule 21). Moving up to that subtree’s parent, the matcher can produce matches to each of rule 2, for the left child’s label as a Reg, and rule 3, for the left child’s label as a T1.

The key to making single operator rules produce the desired results for complex target-machine operations lies in the cost structure of the rule set. With single operator patterns, the sum of the costs of the single operator rules should equal the cost of complex instruction. If the cost of the operation is one cycle and it takes two patterns to match it, then they can be assigned arbitrary non-negative costs that sum to one, such as $\frac{1}{2}$ and $\frac{1}{2}$, or 0 and 1. This approach ensures that a low-cost match generates the instruction sequence with lowest local cost.

The second restriction we impose is that leaves in the tree appear only in singleton rules, such as rules 17 to 20 and 24. Again, the intent is to simplify the tree-pattern matcher. For the situation where a NUM needs to move into a register, rule 18, Reg ← NUM, accomplishes the task. When the NUM can appear as an immediate field, it matches rule 24, T3 ← NUM. Here, T3 is an LHS symbol that only appears in an immediate field of an operation such as addI or loadAI. If the target ISA supported multiple distinct lengths of immediate operands, the rule set would include an LHS symbol for each length.

This second restriction avoids special case code in the tree-pattern matcher. When the matcher considers the children of an operator, it can assume that those children have already been annotated with type and cost information by the singleton rules. The code for operators remains simple and uniform.

Either one or both of these restrictions can be discarded, at the cost of additional complexity in the tree-pattern matcher. In a hand-coded matcher, this simplicity leads to efficient matchers. An automatically-generated, table-driven scheme may have different cost tradeoffs.
11.4.3 Computing Tilings

Given a rule set that maps the compiler’s IR onto the target machine’s ISA, the instruction selector must map a specific AST into an instruction sequence. To do this, a tree-pattern matching code generator constructs a tiling of the AST with the tree-patterns from the rule set. Several techniques for constructing such a tiling exist; they are similar in concept but vary in detail.

To help us understand tiling, consider the AST shown in the margin, a subtree from Figure 11.1.a. It describes a memory operation that loads a word from the address at offset 12 from the label @G. Using the rules from Figure 11.5, we can find a sequence of rules that tile the tree.

Figure 11.7 shows one way to tile this AST. The figure treats the rules as rewrites to the tree. As shown in panel (a), rule 19 matches the LAB node. Rule 19 rewrites the LAB as a Reg, shown in panel (b). Rule 24 matches the NUM node and rewrites it as a T3 to produce the AST shown in panel (c). One of the matches for + (Reg, T3) in panel (c) is rule 9, which rewrites that subtree as a Reg, as shown in panel (d). Rule 5 matches the AST in panel (d) and rewrites it as a single Reg node, not shown. We denote this rewrite sequence as \(\langle 19,24,9,5 \rangle\); order in the sequence matches the order of rule application.

Rules 19 and 24 can be applied in either order. Thus, the sequence \(\langle 19,24,9,5 \rangle\) produces the same tiling as \(\langle 24,19,9,5 \rangle\). The diagram in the margin summarizes those sequences.

This tiling implements the AST; it meets the four criteria defined on page 608. It covers each AST node with one or two operation-tree nodes. The root of each pattern tree overlaps with a leaf in its parent. For example, the root of 19 is a leaf in 9. The connections between patterns are compatible and limited to one node in each subtree. Where two operation-tree nodes cover an AST node, the connected nodes have the same nonterminal label. Thus, this tiling implements the AST.

Given a valid tiling, the selector generates code using the templates associated with each rule. The code template consists of zero or more operations that, together, implement the subtree covered by the rule. For example, the sequence \(\langle 19,24,9,5 \rangle\) implies four operations. Rule 19
11.4 Selection via Tree-Pattern Matching

produces a loadI followed by a loadAI; rule 24 generates no code; rule 9 generates an addI; and rule 5 generates a loadAI with the results of rules 24 and 22. The generated code sequence appears in the margin, with abstract register names to create the correct flow of values. In this case, ⟨24,19,9,5⟩ produces the same code, because rule 24 produces no code. In general, sequences that differ in the order of rule application may generate the operations in a different order.

Choosing Among Multiple Tilings Eight different sequences implement the example AST. Figure 11.8 shows all eight sequences; those that differ only in order appear together. The top row shows the tilings. The middle row shows the code generated by each tiling and its cost. The bottom row shows the sequences for each tiling.

To emit code, the instruction selector must choose one sequence. The obvious choice is to take the low-cost sequence. If the cost estimates for the rules reflect actual runtime costs, then the low-cost sequence should be best. Note that the costs can reflect properties other than execution speed. For example, in an application where code space is critical, a cost metric that reflects the byte-length of the code sequence might be appropriate.

A Tiling Algorithm Figure 11.9 shows a simple recursive algorithm that computes the set of matches for each node in the tree. To keep the exposition simple, the algorithm assumes that the AST and the rule set consist of nodes with either zero, one, or two operands.

\[
\begin{align*}
\text{loadI } \&CP &\Rightarrow r_i \\
\text{loadAI } r_i, @G &\Rightarrow r_j \\
\text{loadAI } r_j, 12 &\Rightarrow r_k
\end{align*}
\]

The sequences that use both rules 18 and 19 would generate the two loadIs in different orders for \langle 18,19 \rangle and \langle 19,18 \rangle.

\[
\begin{align*}
\text{loadI } \&CP &\Rightarrow r_i \\
\text{loadAI } r_i, @G &\Rightarrow r_j \\
\text{addI } r_j, 12 &\Rightarrow r_k \\
\text{load } r_k &\Rightarrow r_l
\end{align*}
\]
**Tile(n)**  
/* n is an AST node */

```plaintext
if n is a leaf then
  Match(n,*) ← { rules that implement n }
else if n is a unary node then
  Tile(child(n))
  Match(n,*) ← ∅  /* Clear n's Match sets */
  for each rule r where operator(r) = operator(n)
    if (child(r),child(n)) are compatible then
      add r to Match(n,class(r))
else if n is a binary node then
  Tile(left(n))
  Tile(right(n))
  Match(n,*) ← ∅  /* Clear n's Match sets */
  for each rule r where operator(r) = operator(n)
    if (left(r),left(n)) and (right(r),right(n)) are compatible then
      add r to Match(n,class(r))
```

**Figure 11.9** Compute All Matches to Tile an AST

This restriction means that the algorithm need only deal with three kinds of tree-patterns: a binary node and its two children, a unary node and its single child, or a leaf node. Limiting the algorithm to this small set of pattern trees keeps the matcher small and efficient.

*Tile* annotates each node with a vector of rules that match the sub-tree rooted at the node. The vector has one element for each LHS symbol in the rule set. By convention and by design, the LHS symbol corresponds to a ⟨storage class,type⟩ pair. For a node *n*, the row `Match(n,*)` contains all of the rules that can implement the subtree rooted at *n*. `Match(n,class)` shows all of the rules that can implement the subtree rooted at *n*, to produce a result in `class`.

For a leaf node, the set of matches can be pre-computed. In our example rule set, a *NUM* node always has the `Match` set shown in the margin, where *ST* is used as an abbreviation for *Stmt*. The rules provide one way to rewrite *NUM* as a *Reg*, with rule 18, and one way to rewrite a *NUM* as a *T3*, with rule 24. Each leaf node has its own vector.

For a unary node *n*, *Tile* first recurs on *n*’s child to discover the ways it can be implemented. It then examines each rule *r* that implements *n*’s operator. If the child has an implementation that is compatible with *r* marked in its `Match` set, then *Tile* adds *r* to `Match(n,class(r))`, where `class(r)` denotes the LHS symbol of rule *r*.

To test compatibility between rule *r* and the implementations for AST node *n*’s child, the algorithm uses a simple test based on the

<table>
<thead>
<tr>
<th>Reg</th>
<th>ST</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td></td>
<td></td>
<td></td>
<td>24</td>
</tr>
</tbody>
</table>

**Match Set for NUM Node**

Should *n* and *r* be algorithm font when used outside of the quotes from the algorithm?
11.4 Selection via Tree-Pattern Matching

Definition of compatibility: the rule and the AST must agree on the operator and on the storage class and value type of the child. Given an AST subtree \( o_1(u) \) and a tree pattern \( o_2(x) \), the two are compatible if and only if (1) the operators in \( o_1 \) and \( o_2 \) are the same; and (2) \( \text{Match}(u, \text{class}(x)) \) is non-empty. The first condition ensures that the tree pattern and the AST subtree use the same operation. The second ensures that the child in the AST can be implemented with a rule that produces the kind of value that rule \( r \) uses.

For a binary node \( n \), \( \text{Tile} \) follows the same plan as for a unary node. It recurs on both the left and right children in the AST. It tests compatibility between the left child in the AST and the corresponding term in rule \( r \) and the right child in the AST and the corresponding term in \( r \). If \( r \) uses the same operator as \( n \) and \( r \) is compatible with the tilings computed for \( n \)'s children, then \( \text{Tile} \) adds \( r \) to \( \text{Match}(n, \text{class}(r)) \).

A Worked Example. Figure 11.10 shows the results of applying \( \text{Tile} \) to our continuing example. Panel (a) shows the AST for \( a \leftarrow b \cdot 2 \times c \). Nodes in the AST are annotated with their postorder numbers; \( \text{Tile} \) traverses the tree in postorder. Panel (d) shows all of the matches that \( \text{Tile} \) finds for each node. Superscripts on the rule numbers indicate the lowest cost for each entry. The low-cost entry in the row is set in **bold**.

To gain a feel for the algorithm, consider the subtree that starts with node 13, which is the example from Figure 11.8. In postorder, \( \text{Tile} \) will compute the \( \text{Match} \) sets for nodes 10 and 11. As leaves, these are pre-computed sets. The only rule that applies to a \( \text{LAB} \) is rule 19, which rewrites \( \text{LAB} \) with \( \text{Reg} \). The table entry reflects that fact. Similarly, the entry for node 11 is the \( \text{Match} \) set shown earlier, which indicates that rule 18 rewrites a \( \text{NUM} \) into a \( \text{Reg} \), and rule 24 rewrites a \( \text{Num} \) into a \( \text{T3} \).

For node 12, \( \text{Tile} \) finds four possibilities. It can produce a \( \text{Reg} \) for the addition, using either rule 8 or rule 9. It can produce a \( \text{T1} \) using rule 21, or it can produce a \( \text{T2} \) using rule 22. As shown in Figure 11.8, each of those choices dictates specific matches for nodes 10 and 11.

Finally, for node 13, \( \text{Tile} \) finds three choices: rules 5, 6, and 7. Each of these rewrites the subtree with a \( \text{Reg} \). A top-down walk in the AST subtree, choosing a compatible set of rules from \( \text{Match} \) will generate the sequences shown in Figure 11.8.

Accounting for Costs. Given the set of matches and the costs for each rule, the compiler can compute the least cost choice in each category, or LHS symbol, during a simple postorder traversal. It accumulates costs for each category, bottom up, by determining the cost for a specific rule choice as the rule’s own cost, plus the cost of the choices at subtrees that the rule requires. The matcher can compute the cost for each rule choice and keep the smallest one for each category.
**Figure 11.10** Results of Running *Tile* on the Low-Level AST for \( a \leftarrow b - 2 \times c \)
11.4 Selection via Tree-Pattern Matching

Tile(n)  /* n is a node in an AST */

if n is a leaf node then
    Match(n,*).rule ← { low-cost rule that matches n, in each class }
    Match(n,*).cost ← { corresponding cost }

else if n is a unary node then
    Tile(child(n))
    Match(n,*).rule ← invalid
    Match(n,*).cost ← largest integer
    for each rule r where operator(r) = operator(n)
        if (child(r), child(n)) are compatible then
            NewCost ← RuleCost(r) + Match(child(n), class(child(r))).cost
            if (Match(n, class(r)).cost > NewCost) then
                Match(n, class(r)).rule ← r
                Match(n, class(r)).cost ← NewCost

else if n is a binary node then
    Tile(left(n))
    Tile(right(n))
    Match(n,*).rule ← invalid
    Match(n,*).cost ← largest integer
    for each rule r where operator(r) = operator(n)
        if (left(r), left(n)) and (right(r), right(n)) are compatible then
            NewCost ← RuleCost(r) + Match(left(n), class(left(r))).cost
            + Match(right(n), class(right(r))).cost
            if (Match(n, class(r)).cost > NewCost) then
                Match(n, class(r)).rule ← r
                Match(n, class(r)).cost ← NewCost

Figure 11.11  Compute Low-Cost Matches to Tile an AST

The cost computation can be folded into Tile, as shown in Figure 11.11. The code is a straightforward extension of Figure 11.9.

The table in Figure 11.10.d shows where, during the matching process, choices occur. Consider node 3. Two patterns match the subtree to produce a value of type Reg. Rule 8 has a total cost of 2: 1 for itself, plus 0 for a Reg at node 1 and 1 for a Reg at node 2. Rule 9 has a total cost of 1: 1 for itself, plus 0 for a Reg at node 1 and 0 for a T3 at node 2. The cost-driven algorithm will keep the match to rule 9. In addition to these matches for Reg, one pattern matches the subtree to produce each of T1 and T2. The low-cost among these matches for node 3 is rule 22, which produces a T2.

By the time the matcher finishes, it has annotated each node with the rule that produces the low-cost match. These matches are shown
in Figure 11.10.b. The instruction selector can then emit code in a bottom-up, postorder pass over the tree. Figure 11.10.c shows the resulting code, with appropriate register names used to tie together the various code templates.

This process yields, at each point, a rule choice that produces a minimal cost code sequence in some local neighborhood in the tree. The literature refers to this property as *local optimality*.

Local optimality does not guarantee that the solution is optimal from a broader perspective, such as the entire procedure or the entire program. In general, compilers cannot efficiently consider all the details that would be necessary to achieve global optimality.

Notice how different the final code is for the three variable references. The store to \texttt{a}, at offset 4 from the ARP, folds completely into a single \texttt{storeAI} operation. The load from \texttt{b}, a call-by-reference parameter whose pointer is 16 bytes before the ARP requires a \texttt{subI} and two \texttt{load} operations; the address-offset address mode only handles positive offsets, and the call-by-reference binding leads to an extra load. Finally, the load from \texttt{c}, a global variable stored at offset 12 from a label, requires a \texttt{loadI} to find the constant pool, followed by two \texttt{loadAI} operations. The first \texttt{loadAI} fetches the address of the label stored at \texttt{@CP + @G}, and the second fetches the value at offset 12 from that address. At the source level, all three references look textually similar; the code generated for them is not.

As a final point, notice that the tree-pattern matching code folds the constant two into the multiply operation. In the example for the peephole selector, the \texttt{loadI} for 2 was not close enough for the simplifier to perform forward substitution. By contrast, the tree-pattern matcher encodes the fact that the subtree has a constant value and carries it along in the set of possible matches to the point where the decision between a \texttt{mult} and a \texttt{multI} occurs.

### 11.4.4 Tools

As we have seen, a tree-oriented, bottom-up approach 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
11.4 Selection via Tree-Pattern Matching

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.

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. If the compiler linearizes the tree into prefix form, it can use a string-matching algorithm 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. Some are faster, some are slower; none is slow enough to have 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 where costs may vary during matching; these systems must perform the dynamic programming during instruction selection. In general, however, all these approaches produce code generators that are both efficient and effective.
SECTION REVIEW

Instruction selection via tree-pattern matching uses trees to represent both operations in the code and operations on the target machine. The compiler writer creates a library of tree patterns that map constructs in the compiler’s IR into operations on the target machine. Each pattern consists of a small IR pattern-tree, a code template, and a cost. In a single pass, the selector finds a locally optimal tiling for the tree. A second postorder walk generates the corresponding code from the templates associated with the tiles.

Several technologies have been used to implement tiling passes. These include hand-coded matchers, 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?

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 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.