Engineering a Compiler

Manuscript for the Third Edition (EaC 3e)

Keith D. Cooper
Linda Torczon

Rice University
Houston, Texas

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 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 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 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 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.

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.

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.

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.

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.

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.

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 information about how to map 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 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.

A back-end generator is sometimes called a *code-generator generator*.

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 are extremely efficient at compile time.

A second key to retargetability is to isolate machine-dependent code to the greatest possible extent. 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. In general, 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 the 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 algorithms for matching tree patterns against trees. The second technique, in Section 11.4, builds on ideas from peephole optimization. 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

The instruction selector must translate from the compiler's final IR into a sequence of operations in the target machine's ISA. The difficulty of that process depends, in any specific compiler, on the form and content of both the IR and the ISA, and on the ambition of the compiler.

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 straighforward, as in translating an ILOC program to run on a simple

Runs Well versus Optimizes Well

When the compiler generates target machine code, it should generate code that will run as quickly as possible. By contrast, when the compiler generates IR in its front end, it should generate code that optimizes as well as possible. A good example of this difference occurs in the way that the compiler encodes an array address polynomial. As we saw in Section 7.5, the naive address polynomial for a reference `A[i,j]` for an array dimensioned as `A[`low_1: $high_1$,low_2: $high_2$`]` in row-major order, would be:

$$\texttt{@A} + (\texttt{i} - low_1) \times len_2 \times w + (\texttt{j} - low_2) \times w.$$

However, in Section 7.5 we recommend instead the polynomial:

$$\begin{aligned}\texttt{@A} &+ (\texttt{i} \times len_2 \times w) + (\texttt{j} \times w) \\ &- (low_1 \times len_2 \times w + low_2 \times w).\end{aligned}$$

While the latter polynomial has more arithmetic operations, it exposes more opportunities for optimization. The final term contains only known constants; the compiler can evaluate it and combine the result with `@A`. The terms based on `i` and `j` are separable; the compiler can move them to different locations where they execute less often. In a typical application, where the source code iterates over consecutive values of `i` and `j`, the polynomial with more operations will optimize to better final code.

In contrast to IR generation, the instruction selector must assume that only limited optimization—scheduling and register allocation—occur after selection. Thus, faced with the same problem, the selector should choose the lower-cost sequence; for precisely this reason, most compilers expand and optimize array references long before instruction selection.

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, 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.

In earlier chapters, we discussed the issues that arise in generating IR from source code. Instruction selection differs from IR generation in two major ways. First, the compiler writer has different goals for IR generaton than for instruction selection; 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—any given processor might provide multiple ways to implement an IR construct, each with its own costs and its own restrictions. When the code generator finds multiple ways to implement a given IR construct, it needs to choose among them based on knowledge of the operations' costs and the construct's 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 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.

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.

While we discuss these problems in terms of a processor that runs ILOC, remember that ILOC is an extremely simple case.

Duplicate Implementations 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 operation $\mathtt{r}_i \rightarrow \mathtt{r}_j$, that copies the value in $\mathtt{r}_i$ into $\mathtt{r}_j$. 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 $\mathtt{r}_i \rightarrow \mathtt{r}_j$ 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,

`addI` r_i`,0` $\Rightarrow r_j$	`subI` r_i`,0` $\Rightarrow r_j$	`multI` r_i`,1` $\Rightarrow r_j$
`divI` r_i`,1` $\Rightarrow r_j$	`lshiftI` r_i`,0` $\Rightarrow r_j$	`rshiftI` r_i`,0` $\Rightarrow r_j$
`and` r_i`,`r_i $\Rightarrow r_j$	`orI` r_i`,0` $\Rightarrow r_j$	`xorI` r_i`,0` $\Rightarrow r_j$

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

A programmer would discount most, if not all, of these alternate sequences. Using 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 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 a copy operation.

Address Modes The arithmetic operations shown above all assumed a register-to-register model. Some operations must address memory; loads and stores are classic examples. To make common address computations efficient, processors include address modes on load and store operations that capture common assembly-language idioms.

In ILOC, `load` and `store` take register addresses. Other forms take a register-based address and a register-based offset (`loadAO` and `storeAO`) or a register-based address and an immediate offset (`loadAI` and `storeAI`). Typically, such address modes lead to more efficient code by avoiding an explicit operation for the arithmetic and the need for a register to hold the addition's result. The address-mode arithmetic is performed inside the load-store unit, so that its use may free up a general-purpose functional unit for other calcuations.

Arithmetic operations in ILOC have just two address modes: a three-register mode and an immediate mode. In three-register mode, two arguments and one result all reside in independently specified registers. In immediate mode, one of those arguments is a known constant value specified in the text of the operation. Real processors may provide (1) address modes that allow arithmetic operations to either read arguments from or write results to memory locations; (2) one or two address operations that overwrite one or more of the input arguments; or (3) operations with implict arguments, such as the top of a runtime stack. Each of those situations creates more options that the instruction selector must consider.

PC-relative branch: A transfer of control that specifies an offset, either positive or negative, from its own memory address.

Control-flow operations have similar issues. 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.

Real Processors Real processors are more complicated than ILOC. They may provide operations at several levels of abstraction. For example:

- A string-move operation allows the code to easily specify a complex sequence that might includes a loop, combining multiple assignments into one move opeation.
- 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.

Processors place idiosyncratic constraints on register use. An integer multiply might need to take its operands from a specific subrange of the register set. A double-precision floating-point operation might need its operands in even-numbered register pairs. A memory operation might only execute on one of the processor's functional units.

Destructive operation: A two-address operation that overwrites one of its operands is a *destructive* operation.

Some processors provide *destructive* two-address operations. In mapping from a three-address IR to a two-address, destructive operation, the code generator must account for the fact that the operation kills one of its operands. It may swap the order of the operands to a commutative, destructive operation to control which value is killed.

Features such as these complicate instruction selection, because they add context, they add new complexities, and they add choices. Automated techniques for building instruction selection arose, in large part, as a response to increasing processor complexity. By specifying 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.

Costs Each operation 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, such as load, store, 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. The latency of multiplication and division may depend on the bit-patterns in the operands.

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.

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

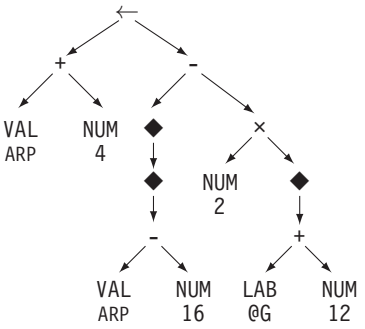
While instruction selection can play an important role in determining code quality, the compiler writer must keep in mind the enormous size of the search space that the instruction selector might explore. Even moderately-sized instruction sets can produce search spaces that contain millions of states. 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.

Motivating Example

The discussions of tree-pattern matching and peephole optimization 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 × `c`. Figure 11.1 shows the example in two different IRs. Panel (a) shows a low-level AST for the statement; the discussion of tree-pattern matching uses this IR. Panel (b) shows the same statement in quadruple form; the discussion of peephole-based instruction selection starts from this IR.

This same example appears in Figure 5.1 on page 230.

The example uses three variables. `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 that resides at offset 12 from the global label `@G`. This variety of storage locations will help highlight the kinds of code that the compiler will generate for references.



(a) Low-Level AST

Op	Arg_1	Arg_2	Result
×	2	c	t_1
-	b	t_1	a

(b) Quadruples

FIGURE 11.1 Low-Level IRs for a ←b - 2 × c

Low-Level AST To expose enough detail for instruction selection, the AST shown in Figure 11.1.a 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., multI).
- 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.

Constant pool: A data area set aside for large constant values, so that they can be loaded on demand. The constant pool is, typically, statically initialized.

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 multI 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.

Quadruples Figure 11.1.b 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. As we will see in Section 11.4, the peephole instruction selector will immediately expand this code to ensure that the needed detail explicit and exposed.

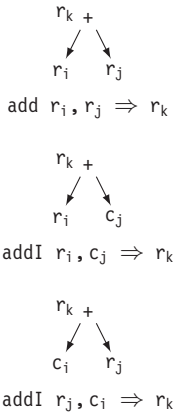
SECTION REVIEW

Need to write a summary that actually relates to the new text.

REVIEW QUESTIONS

Need new review questions.

11.3 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 a tree-pattern matching problem, 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 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 tiling is a set of $\langle x,y \rangle$ pairs, where x is a node in the AST and y is the root of an operation tree. 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.

A tiling *implements* the AST if and only if:

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

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.

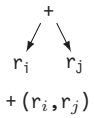
Trees versus DAGs

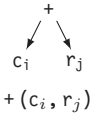
Tree-pattern matching operates, as the name implies, on trees. If the compiler has constructed DAGs at some point prior to instruction selection, it needs to convert the DAGs back into tree form. The compiler could simply create duplicate copies of the shared subtrees, one for each parent of a shared subtree. That approach would discard any benefits, such as eliminating redundant computations, of the DAG.

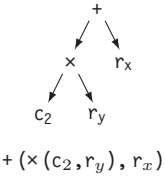
The alternative is to reshape the DAG so that it considers each node that has multiple ancestors as the root of a tree—creating a forest from the DAG with an implicit order of evaluation among the trees specified by the original evaluation order in the DAG. Results are shared by giving them compiler-generated names and connecting their definition, at the root of a new tree, with their uses, at new leaf nodes that replace the shared subtrees.

11.3.1 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 $+(r_i,r_j)$ in prefix form, and the `addI` operation is $+(c_i,r_j)$. 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 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: $+(\times(c_2,r_y),r_x)$. Leaves are assigned symbolic names that encode information about the type and storage location of the operand. For example, r_i indicates a value that resides in a register and c_j indicates a generic constant value. In an AST, labels such as `VAL`, `NUM`, `LAB`, and `CON` provide more detailed information (See page 618). Subscripts are added to names for uniqueness. If we rewrite the AST from Figure 11.1.a in prefix form, it becomes:

$$\leftarrow\ (+(\mathtt{VAL}_1,\mathtt{NUM}_1),\ -(\blacklozenge(\blacklozenge(-(\mathtt{VAL}_2,\mathtt{NUM}_2))),\times(\mathtt{NUM}_3,\blacklozenge(+(\mathtt{LAB}_1,\mathtt{NUM}_4)))))$$

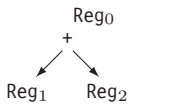
While the drawing of the tree may be more intuitive, this linear prefix form contains precisely the same information.

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).

11.3.2 Rewrite Rules

To build an instruction selector based on tree-pattern matching, the compiler writer must design a set of rewrite rules for the AST 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 11.2 shows the ILOC subset that we will us in the ongoing examples.

We will structure a collection of tree patterns into a set of rewrite rules. The selector will look for a subtree that matches the right-hand side (RHS) of a pattern; such a match indicates that it can rewrite the subtree with the symbol from the pattern's left-hand side (LHS). Some symbols appear only on the RHS of a pattern; these represent concrete symbols that can be leaves in a tree. These symbols are somewhat analogous to terminal symbols in a context-free grammar. Other symbols appear on either the LHS or RHS of a pattern; these symbols are analogous to nonterminal symbols in a context-free grammar.



$Reg_0 \rightarrow +(Reg_1, Reg_2)$

Figure 11.3 shows a set of rewrite rules to tile our low-level AST with ILOC operations. Each rule represents a small AST. For example, rule 8 describes the familiar tree shown in the margin. The rule's RHS consists of the + node and its two children, while the label on the + node, Reg_0, 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 accomodate 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 `@CP`. (In practive, `@CP` would undoubtedly be constructed by mangling the name of the procedure.) Further, it assumes that `@L` is the positive offset of Lab_1 from `@CP`. (In practice, the instruction selector would need to lookup Lab_1 in a table to find the value of `@L`.) With

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 11.2 The ILOC Subset

	Production	Cost	Code Template
0	Goal → Goal Stmt	0	
1	Goal → Stmt	0	
2	Stmt → ← (Reg$_1$,Reg$_2$)	3	store r$_2$ ⇒ r$_1$
3	Stmt → ← (T1$_1$,Reg$_2$)	3	storeAO r$_2$ ⇒ T1.r$_1$,T1.r$_2$
4	Stmt → ← (T2$_1$,Reg$_2$)	3	storeAI r$_2$ ⇒ T2.r,T2.n
5	Reg → ◆ (Reg$_1$)	3	load r$_1$ ⇒ r$_{new}$
6	Reg → ◆ (T1$_1$)	3	loadAO T1.r$_1$,T1.r$_2$ ⇒ r$_{new}$
7	Reg → ◆ (T2$_1$)	3	loadAI T2.r,T2.n ⇒ r$_{new}$
8	Reg → + (Reg$_1$,Reg$_2$)	1	add r$_1$,r$_2$ ⇒ r$_{new}$
9	Reg → + (Reg$_1$,T3$_2$)	1	addI r$_1$,T3 ⇒ r$_{new}$
10	Reg → + (T3$_1$,Reg$_2$)	1	addI r$_2$,T3 ⇒ r$_{new}$
11	Reg → - (Reg$_1$,Reg$_2$)	1	sub r$_1$,r$_2$ ⇒ r$_{new}$
12	Reg → - (Reg$_1$,T3$_2$)	1	subI r$_1$,T3 ⇒ r$_{new}$
13	Reg → - (T3$_1$,Reg$_2$)	1	rsubI r$_2$,T3 ⇒ r$_{new}$
14	Reg → × (Reg$_1$,Reg$_2$)	2	mult r$_1$,r$_2$ ⇒ r$_{new}$
15	Reg → × (Reg$_1$,T3$_2$)	2	multI r$_1$,T3 ⇒ r$_{new}$
16	Reg → × (T3$_1$,Reg$_2$)	2	multI r$_2$,T3 ⇒ r$_{new}$
17	Reg → CON$_1$	1	loadI C$_1$ ⇒ r$_{new}$
18	Reg → NUM$_1$	1	loadI N$_1$ ⇒ r$_{new}$
19	Reg → LAB$_1$	4	loadI @CP ⇒ r$_{new_1}$ loadAI r$_{new_1}$,@L ⇒ r$_{new_2}$
20	Reg → VAL$_1$	0	
21	T1 → + (Reg$_1$,Reg$_2$)	0	
22	T2 → + (Reg$_1$,T3$_2$)	0	
23	T2 → + (T3$_1$,Reg$_2$)	0	
24	T3 → NUM$_1$	0	

FIGURE 11.3 Rewrite Rules for Tiling the Low-Level Tree with ILOC

these assumptions, the emitted code loads @CP into a register and uses it as the base address for a loadAI operation.

The rule set in Figure 11.3 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 Stmt, the non-terminal symbol for an assignment statement.

Interactions between the patterns, encoded through the use of LHS symbols, define the ways in which subtrees can combine. For example, productions 5 through 20 each have Reg on their LHS thus, each of those rules describes a subtree that can rewrite a Reg.

The LHS symbols encode knowledge about the type and storage class of the values that they represent. For example, a Reg represents a value in a register. A Reg might result from a subtree that produces an integer value stored in a register, as in rules 5 through 16. It might also result from a CON, a NUM, or a LAB, through productions 17, 18, and 19. Rule 20 provides a way to convert a VAL node to a register—effectively, 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.

To keep the examples simple, we assume that a LAB can be loaded with a loadI. In practice, the LAB would probably generate a loadAI from a data area for labels and large constants.

T1 and T2 represent addresses—value that can be computed in address modes such as loadAO 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 add operation, while rule 22 represents an add operation done by the memory addressing hardware.

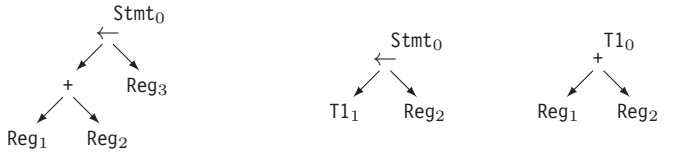
Some matching techniques only allow fixed costs. Others allow the costs to vary during matching to reflect prior choices.

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 11.3 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 the low-cost option among possible alternatives.

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 operatiosn to handle more complex cases.

Restrictions on the Rule Set We restricted the rules in Figure 11.3 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 11.3.3 without changing its expressiveness.

To see this, consider two cases shown in Figure 11.4. Panel (a)



(a) Single Pattern for `storeAO` (b) Single-Operator Patterns for `storeAO`

FIGURE 11.4 Single-operator patterns versus multi-operator patterns

shows a single pattern for ← (+ (Reg,Reg),Reg). 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 special case matters if the hardware can compute the sum in the addressing unit, as in `storeAO`.

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 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. When the compiler writer uses a set of single operator patterns to produce a complex instruction, the sum of the costs of the single operator rules should equal the cost of instruction. If the cost of the operation is one cycle and it takes two patterns to match it, then they can be assigned costs abritrary 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

Notice that rule 18 has a code template, but rule 24 does not.

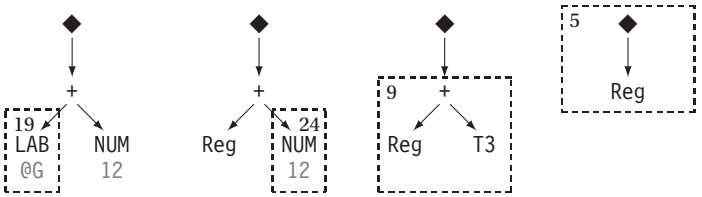


FIGURE 11.5 A Simple Tree Rewrite Sequence

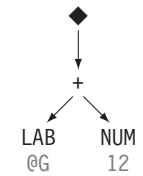
ISA supported multiple distinct lengths of immediate operands, the rule set would include an LHS symbol for each length.

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.

11.3.3 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.



Example Subtree

To help us understand tiling, consider the AST shown in the margin, a subtree of 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 11.3, we can find a sequence of rules that tile the tree.

Figure 11.5 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 ⟨19,24,9,5⟩; order in the sequence matches the order of rule application.

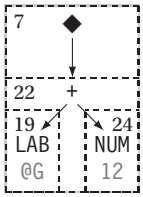
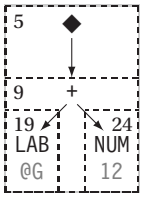
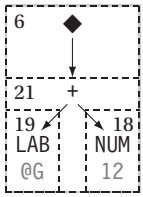
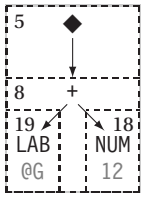
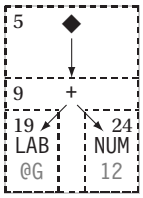
Tiling				
Code	`loadI @CP ⇒ r`$_i$ `loadAI r`$_i$`,@G ⇒ r`$_j$ `loadAI r`$_j$`,12 ⇒ r`$_k$ Cost: 7 cycles	`loadI @CP ⇒ r`$_i$ `loadAI r`$_i$`,@G ⇒ r`$_j$ `addI r`$_j$`,12 ⇒ r`$_k$ `load r`$_k$ `⇒ r`$_l$ Cost: 8 cycles	`loadI @CP ⇒ r`$_i$ `loadAI r`$_i$`,@G ⇒ r`$_j$ `loadI 12 ⇒ r`$_k$ `loadAO r`$_j$`,r`$_k$ `⇒ r`$_l$ Cost: 8 cycles	`loadI @CP ⇒ r`$_i$ `loadAI r`$_i$`,@G ⇒ r`$_j$ `loadI 12 ⇒ r`$_k$ `add r`$_j$`,r`$_k$ `⇒ r`$_l$ `load r`$_l$ `⇒ r`$_m$ Cost: 9 cycles
Sequences	⟨19,24,22,7⟩ ⟨24,19,22,7⟩	⟨19,24,9,5⟩ ⟨24,19,9,5⟩	⟨19,18,21,6⟩ ⟨18,19,21,6⟩	⟨19,18,8,5⟩ ⟨18,19,8,5⟩

FIGURE 11.6 The Set of All Tilings for the Example Subtree

Rules 19 and 24 can be applied in either order. Thus, the sequence ⟨19,24,9,5⟩ produces the same tiling as ⟨24,19,9,5⟩. The diagram in the margin summarizes those sequences.



⟨19,24,9,5⟩
⟨24,19,9,5⟩

This tiling implements the AST; it meets the four criteria defined on page 620. It covers each AST node with one or two operation-tree nodes. The four pattern trees connect; the root of each pattern tree overlaps with a leaf in its parent. For example, the root of 19 is a leaf in 9. Finally, the connections are compatible. 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 that the rule covers. For example, the low-cost sequence ⟨19,24,22,7⟩ produces a three operation sequence. Rule 19 generates a `loadI` followed by a `loadAI`; rules 24 and 22 generate no code; and rule 7 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, ⟨24,19,22,7⟩ 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.

`loadI @CP ⇒ r`$_i$
`loadAI r`$_i$`,@G ⇒ r`$_j$
`loadAI r`$_j$`,12 ⇒ r`$_k$

```
Tile(n)     /* n is an AST node */
  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 Labels 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))
```

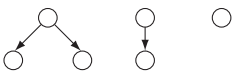
FIGURE 11.7 Compute All Matches to Tile an AST

Choosing Among Multiple Tilings Eight different sequences implement the example AST. Figure 11.6 shows all eight the 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 two sequences that generate the tiling.

The sequences that use both rules 18 and 19 would generate the two `loadI`s in different orders for ⟨18,19⟩ and ⟨19,18⟩.

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 11.7 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. 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 subtree rooted at the node. The vector has one element for each LHS sym-

bol 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.

Reg	ST	T1	T2	T3
18				24

Match Set for NUM Node

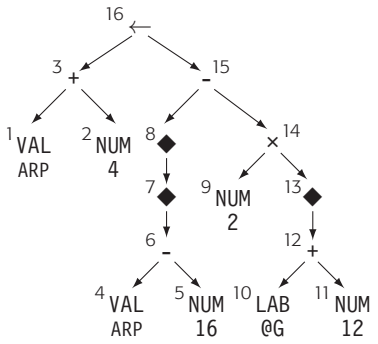
For a unary node *n*, *Tile* first recurs on *n*'s child to discover the ways it can be implemented. It then examines each rule *r* that implements *n*'s operator. If the child has an implementation, marked in its *Match* set, that is compatible with *r*, then *Tile* adds *r* to *Match(n,class(r))*,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

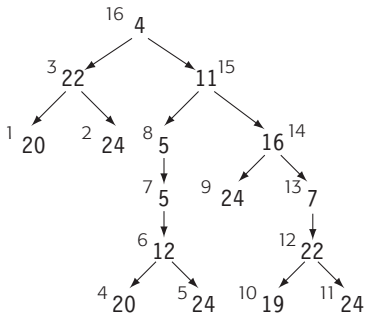
For a binary node *n*, *Tile* follows the same plan as for a unary node. It recurs on both the left and right children in the AST. It tests compatibility between the left child in the AST and the corresponding term in rule *r* and the right child in the AST and the corresponding term in *r*. If *r* uses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

A Worked Example Figure 11.8 shows the results of applying *Tile* to our continuing example. Panel (a) shows the AST for a ←b – 2 × c. Nodes in the AST are annotated with their postorder numbers; *Tile* traverses the tree in postorder. Panel (d) shows all of the matches that *Tile* finds for each node. Superscripts on the rule numbers indicate the cost of each match. The low-cost match is printed in **bold**.

To gain a feel for the algorithm, consider the subtree that starts with node 13, which is just our example from Figure 11.6. In postorder, *Tile* will compute the *Match* sets for node 10 and then node 11. As leaves, these are precomputed sets. The only rule that applies to a LAB is rule 19, which rewrites LAB with Reg. The table entry reflects that fact. Similarly, the entry for node 12 is the *Match* set shown earlier, which



(a) Low-Level AST for a ← b - 2 × c



(b) Top-down Assignment of Rules

Node	Rule	Emitted Code			
6	12	subI	r_{arp}, 16	⇒	r_a
7	5	load	r_a	⇒	r_b
8	5	load	r_b	⇒	r_c
10	19	loadI	@CP	⇒	r_d
		loadAI	r_d, @G	⇒	r_e
13	7	loadAI	r_e, 12	⇒	r_f
14	16	multI	r_f, 2	⇒	r_g
15	11	sub	r_c, r_g	⇒	r_h
16	4	storeAI	r_h	⇒	r_{arp}, 4

(c) Code Emmited by Rules

Node	Reg	Stmt	T1	T2	T3
1	**20** 0				
2	18 1				**24** 0
3	8 2 9 1		21 1	**22** 0	
4	**20** 0				
5	18 1				**24** 0
6	11 2 **12** 1				
7	**5** 4				
8	**5** 7				
9	18 1				**24** 0
10	**19** 4				
11	18 1				**24** 0
12	8 6 9 5		21 5	**22** 4	
13	5 8 6 8 **7** 7				
14	14 10 **16** 9				
15	**11** 17				
16		2 21 3 21 **4** 20			

(d) Full Set of Matches and Costs

FIGURE 11.8 Results of Running *Tile* on the Low-Level AST for a ←b - 2 × c

indicates that rule 18 rewrites a NUM into a Reg, and rule 24 rewrites a Num into a T3.

Moving to node 12, *Tile* finds four possibilites. It can produce a Reg for the addition, using either rule 8 or rule 9. It can produce a T1 using rule 21, or it can produce a T2 using rule 22. As shown in Figure 11.6, each of those choices dictates specific matches for nodes 10 and 11.

Finally, for node 13, *Tile* finds three choices: rules 5, 6, and 7. Each of these rewrites the subtree with a Reg. A top-down walk in the AST subtree, choosing a compatible set of rules from *Match* will generate the sequences shown in Figure 11.6.

```
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 11.9 Compute Low-Cost Matches to Tile an AST

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, suring 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.

The cost computation can be folded into *Tile*, as shown in Figure 11.9. The code is a straightforward extension of Figure 11.7.

The table in Figure 11.8.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 11.8.b. The instruction selector can then emit code in a bottom-up, postorder pass over the tree. Figure 11.8.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: A scheme in which the compiler has no better alternative, at each point in the code, is considered *locally optimal*.

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, we expect that compilers cannot efficiently consider all the details that would be necessary to reason about global optimality.

As a final point from the example, notice how different the final code is for the three variable references. The store to `a`, at offset 4 from the ARP, folds completely into a single `storeAI` operation. The load from `b`, a call-by-reference parameter whose pointer is 16 bytes before the ARP requires a `subI` and two `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 `c`, a global variable stored at offset 12 from a label, requires a `loadI` to find the constant pool, followed by two `loadAI` operations. The first `loadAI` fetches the address of the label stored at `@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.

11.3.4 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. 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 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 pattern-tree, a code template, and a cost. In a single pass, the selector can find a locally optimal tiling for the tree. A second postorder walk can generate the corresponding code from the templates associated with the tiles.

Several technologies have been used to implement tiling passes. These include hand-coded matchers such as the one shown in Figure 11.7, 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?