2.12 How Compilers Work: An Introduction

The purpose of this section is to give a brief overview of the compiler function, which will help the reader understand both how the compiler translates a high-level language program into machine instructions. Keep in mind that the subject of compiler construction is usually taught in a one- or two-semester course; our introduction will necessarily only touch on the basics.

Our description of the functions of a compiler follows the structure in Figure 2.31 on page 116. To illustrate the concepts in this section, we will use the C version of a *while* loop from an earlier section:

```
while (save[i] == k)
    i += 1;
```

The Front End

The function of the front end is to read in a source program, check the syntax and semantics, and translate the source program to an intermediate form that interprets most of the language-specific operation of the program. As we will see, intermediate forms are usually simple, and some are in fact similar to the Java bytecodes, which can be found in Section 2.14.

The front end is usually broken into four separate functions:

1. *Scanning* reads in individual characters and creates a string of tokens. Examples of *tokens* are reserved words, names, operators, and punctuation symbols. In the above example, the token sequence is "while", "(", "save", "[", "i", "]", "==", "k", ")", "i", "+=", "1". A word like "while" is recognized as a reserved word in C, but "save", "i", and "j" are recognized as names, and "1" is recognized as a number.

2. *Parsing* takes the token stream, ensures the syntax is correct, and produces an *abstract syntax tree*, which is a representation of the syntactic structure of the program. Figure 2.12.1 shows what the abstract syntax tree might look like for this program fragment.

3. *Semantic analysis* takes the abstract syntax tree and checks the program for semantic correctness. Semantic checks normally ensure that variables and types are properly declared and that the types of operators and objects match, a step called *type checking*. During this process, a symbol table representing all the named objects—classes, variables, and functions—is usually created and used to type check the program.

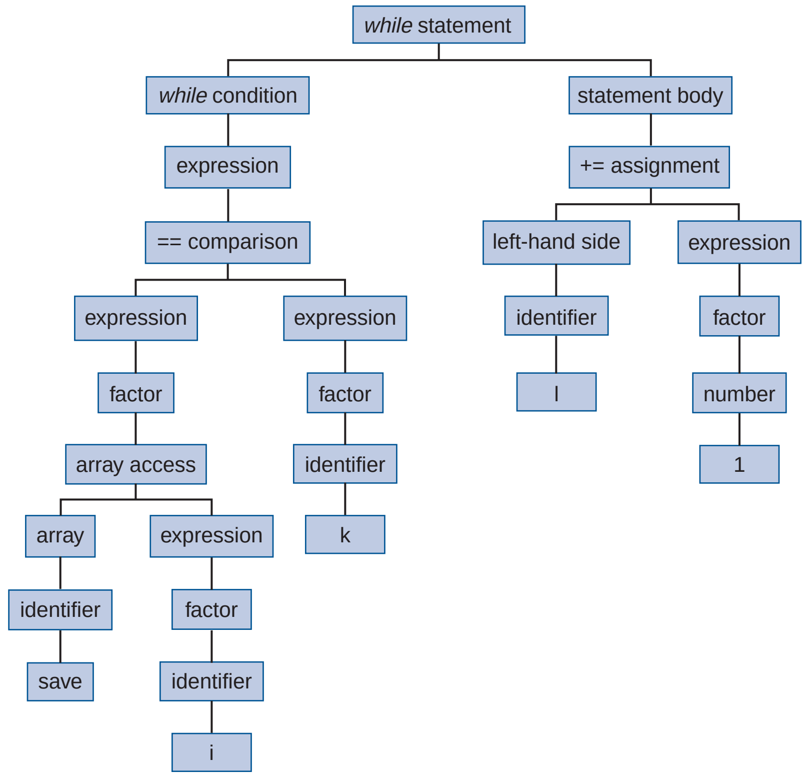

FIGURE 2.12.1 An abstract syntax tree for the *while* example. The roots of the tree consist of the informational tokens such as numbers and names. Long chains of straight-line descendents are often omitted in constructing the tree.

4. *Generation of the intermediate representation* (IR) takes the symbol table and the abstract syntax tree and generates the intermediate representation that is the output of the front end. Intermediate representations usually use simple operations on a small set of primitive types, such as integers, characters, and reals. Java bytecodes represent one type of intermediate form. In modern compilers, the most common intermediate form looks much like the MIPS instruction set but with an infinite number of virtual registers; later, we describe how to map these virtual registers to a finite set of real registers. Figure 2.12.2 shows how our example might be represented in such an intermediate form. We capitalize the MIPS instructions in this section when they represent IR forms.

```
        # comments are written like this--source code often included
        # while (save[i] == k)
loop:   LI R1,save     # loads the starting address of save into R1
        LW R2,i
        MULT R3,R2,4 #Multiply R2 by 4
        ADD R4,R3,R1
        LW R5,0(R4) # load save[i]
        LW R6,k
        BNE R5,R6,endwhileloop
        # i += 1
        LW R6, i
        ADD R7,R6,1  # increment
        SW R7,i
        branch loop # next iteration
endwhileloop:
```

FIGURE 2.12.2 The *while* loop example is shown using a typical intermediate representation. In practice, the names `save, i, k` would be replaced by some sort of address such as a reference to either the local stack pointer or a global pointer and an offset, similar to the way `save[i]` is accessed. Note that the format of the MIPS instructions is different because they represent intermediate representations here: the operations are capitalized and the registers use `RXX` notation.

The intermediate form specifies the functionality of the program in a manner independent of the original source. After this front end has created the intermediate form, the remaining passes are largely language independent.

Implementing Local Optimizations

Local optimizations are implemented on basic blocks by scanning the basic block in instruction execution order looking for optimization opportunities. In the assignment statement example on page 116 in the prior Section 2.11, the duplication of the entire address calculation is recognized by a series of sequential passes over the code. Here is how the process might proceed, including a description of the checks that are needed:

1. Determine that the two `li` operations return the same result by observing that the operand `x` is the same and that the value of its address has not been changed between the two `li` operations.
2. Replace all uses of `R106` in the basic block by `R101`.

3. Observe that `i` cannot change between the two `LW`s that reference it. So replace all uses of `R107` by `R101`.
4. Observe that the `mult` instructions now have the same input operands, so that `R108` may be replaced by `R102`.
5. Observe that now the two `add` instructions have identical input operands (`R100` and `R102`), so replace the `R109` by `R103`.
6. Use dead store code elimination to delete the second set of `li`, `lw`, `mult`, and `add` instructions since their results are unused.

Throughout this process, we need to know when two instances of an operand have the same value. This is easy to determine when they refer to virtual registers, since our intermediate representation uses such registers only once, but the problem can be trickier when the operands are variables in memory, even though we are only considering references within a basic block.

It is reasonably easy for the compiler to make the common subexpression elimination determination in a conservative fashion in this case; as we will see in the next subsection, this is harder when branches intervene.

Implementing Global Optimizations

To understand the challenge of implementing global optimizations, let's consider a few examples:

- Consider the case of an opportunity for common subexpression elimination, say, of an IR statement like `ADD Rx,R20,R50`. To determine whether two such statements compute the same value, we must determine whether the values of `R20` and `R50` are identical in the two statements. In practice, this means that the values of `R20` and `R50` have not changed between the first statement and the second. For a single basic block, this is easy to decide; it is more difficult for a more complex program structure involving multiple basic blocks and branches.
- Consider the second `LW` of `i` into `R107` within the earlier example on page 116: how do we know whether its value is used again? If we consider only a single basic block and we know that all uses of `R107` are within that block, it is easy to see. As optimization proceeds, however, common subexpression elimination and copy propagation may create other uses of a value. Determining that a value is unused and the code is dead is more difficult in the case of multiple basic blocks.
- Finally, consider the load of `k` in our loop, which is a candidate for code motion. In this simple example, we might argue it is easy to see that `k` is not changed in the loop and is, hence, loop invariant. Imagine, however, a more complex loop with multiple nestings and *if* statements within the body. Determining that the load of `k` is loop invariant is harder in such a case.

The information we need to perform these global optimizations is similar: we need to know where each operand in an IR statement could have been changed or *defined* (use-definition information). The dual of this information is also needed: that is, finding all the uses of that changed operand (definition-use information). *Data flow analysis* obtains both types of information.

Global optimizations and data flow analysis operate on a *control flow graph*, where the nodes represent basic blocks and the arcs represent control flow between basic blocks. Figure 2.12.3 shows the control flow graph for our simple loop example, with one important transformation introduced. We describe the transformation in the caption, but see if you can discover it, and why it was done, on your own!

Suppose we have computed the use-definition information for the control flow graph in Figure 2.12.3. How does this information allow us to perform code motion? Consider IR statements number 1 and 6: in both cases the use-definition information tells us that there are no definitions (changes) of the operands of these statements within the loop. Thus, these IR statements can be moved outside the loop. Notice that if the `LI` of `save` and the `LW` of `k` are executed once, just prior to the loop entrance, the computational effect is the same, but the program now runs faster since these two statements are outside the loop. In contrast, consider IR statement 2, which loads the value of `i`. The definitions of `i` that affect this statement are both outside the loop,

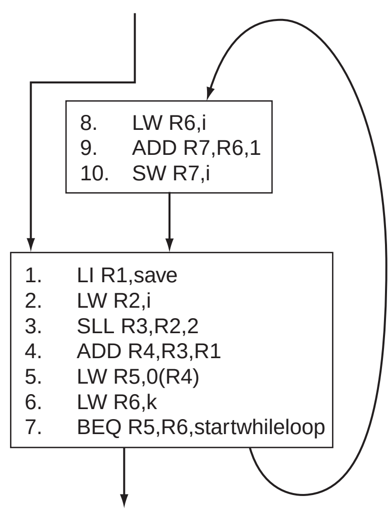

FIGURE 2.12.3 A control flow graph for the *while* loop example. Each node represents a basic block, which terminates with a branch or by sequential fall-through into another basic block that is also the target of a branch. The IR statements have been numbered for ease in referring to them. The important transformation performed was to move the *while* test and conditional branch to the end. This eliminates the unconditional branch that was formerly inside the loop and places it before the loop. This transformation is so important that many compilers do it during the generation of the IR. The `MULT` was also replaced with ("strength-reduced to") a `SLL`.

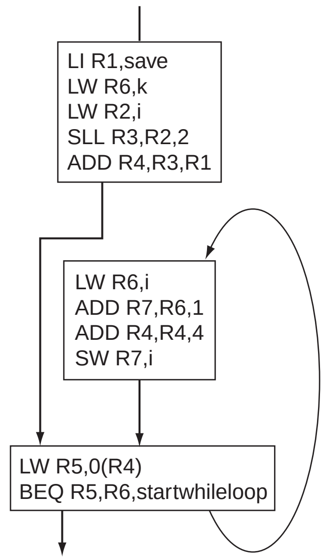

FIGURE 2.12.4 The control flow graph representation of the *while* loop example after code motion and induction variable elimination. The number of instructions in the inner loop has been reduced from 10 to 6.

where `i` is initially defined, and inside the loop in statement 10 where it is stored into. Hence, this statement is not loop invariant. Figure 2.12.4 shows the code after performing both code motion and induction variable elimination, which simplifies the address calculation. The variable `i` can still be register allocated, eliminating the need to load and store it every time, and we will see how this is done in the next subsection.

Before we turn to register allocation, we need to mention a caveat, which also illustrates the complexity and difficulty of optimizers. Remember that the compiler must be conservative. To be conservative, a compiler must consider the following question: Is there *any way* that the variable `k` could possibly ever change in this loop? Unfortunately, there is one. Suppose that the variable `k` and the variable `i` actually refer to the same memory location, which could happen if they were accessed by pointers or reference parameters.

I am sure that many readers are saying, "Well, that would certainly be a stupid piece of code!" Alas, this response is not open to the compiler, which must translate the code as it is written. Recall also that the aliasing information must also be conservative; thus, compilers often find themselves negating optimization opportunities because of a possible alias that exists in one place in the code or because of incomplete information about aliasing.

Register Allocation

Register allocation is perhaps the most important optimization for modern load-store architectures. Eliminating a load or a store eliminates an instruction. Furthermore, register allocation enhances the value of other optimizations, such as common subexpression elimination. Fortunately, the trend toward larger register counts in modern architectures has made register allocation simpler and more effective. Register allocation is done on both a local basis and a global basis, that is, across multiple basic blocks but within a single function. Local register allocation is usually done late in compilation, as the final code is generated. Our focus here is on the more challenging and more opportunistic global register allocation.

Modern global register allocation uses a region-based approach, where a region (sometimes called a *live range*) represents a section of code during which a particular variable could be allocated to a particular register. How is a region selected? The process is iterative:

1. Choose a definition (change) of a variable in a given basic block; add that block to the region.
2. Find any uses of that definition, which is a data flow analysis problem; add any basic blocks that contain such uses, as well as any basic block that the value passes through to reach a use, to the region.
3. Find any other definitions that also can affect a use found in the previous step and add the basic blocks containing those definitions, as well as the blocks the definitions pass through to reach a use, to the region.
4. Repeat steps 2 and 3 using the definitions discovered in step 3, until convergence.

The set of basic blocks found by this technique has a special property: if the designated variable is allocated to a register in all these basic blocks, then there is no need for loading and storing the variable.

Modern global register allocators start by constructing the regions for every virtual register in a function. Once the regions are constructed, the key question is how to allocate a register to each region: the challenge is that certain regions overlap and may not use the same register. Regions that do not overlap (i.e., share no common basic blocks) can share the same register. One way to represent the interference among regions is with an *interference graph,* where each node represents a region and the arcs between nodes represent that the regions have some basic blocks in common.

Once an interference graph has been constructed, the problem of allocating registers is equivalent to a famous problem called *graph coloring:* find a color for each node in a graph such that no two adjacent nodes have the same color. If the number of colors equals the number of registers, then coloring an interference graph is equivalent to allocating a register for each region! This insight was the initial motivation for the allocation method now known as region-based allocation, but originally called the graph-coloring approach. Figure 2.12.5 shows the flow graph representation of the *while* loop example after register allocation.

What happens if the graph cannot be colored using the number of registers available? The allocator must spill registers until it can complete the coloring. By doing the coloring based on a priority function that takes into account the number of memory references saved and the cost of tying up the register, the allocator attempts to avoid spilling for the most important candidates. Spilling is equivalent to splitting up a region (or live range); if the region is split, fewer other regions will interfere with the two separate nodes representing the original region. A process of splitting regions and successive coloring is used to allow the allocation process to complete, at which point all candidates will have been allocated a register. Of course, whenever a region is split, loads and stores must be introduced to get the value from memory or to store it there. Choosing the location to split a region must balance the cost of the loads and stores that must be introduced against the advantage of freeing up a register and reducing the number of interferences.

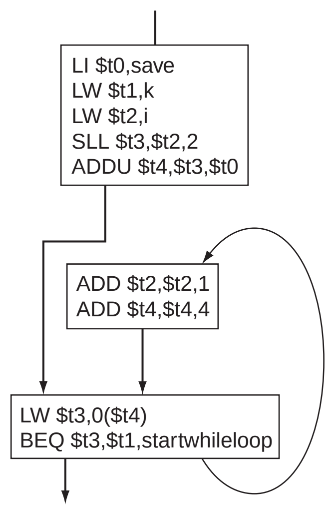

FIGURE 2.12.5 The control flow graph representation of the *while* loop example after code motion and induction variable elimination and register allocation, using the MIPS register names. The number of IR statements in the inner loop has now dropped to only 4 from 6 before register allocation and 10 before any global optimizations. The value of `i` resides in `$t2` at the end of the loop and may need to be stored eventually to maintain the program semantics. If `i` were unused after the loop, not only could the store be avoided, but in fact the increment inside the loop could be eliminated completely!

Modern register allocators are incredibly effective in using the large register counts available in modern processors. In many programs, the effectiveness of register allocation is limited not by the availability of registers but by the possibilities of aliasing that cause the compiler to be conservative in its choice of candidates.

Code Generation

The final steps of the compiler are code generation and assembly. Most compilers do not use a stand-alone assembler that accepts source; to save time, they instead perform most of the same functions: filling in symbolic values and generating the binary code as the final stage of code generation.

In modern processors, code generation is reasonably straightforward, since the simple architectures make the choice of instruction relatively obvious. For more complex architectures, such as the IA-32, code generation is more complex since multiple IR instructions may collapse into a single machine instruction. In modern compilers, this compilation process uses pattern matching with either a tree-based pattern matcher or a pattern matcher driven by a parser.

During code generation, the final stages of machine-dependent optimization are also performed. These include some constant folding optimizations, as well as localized instruction scheduling (see Chapter 6).

Elaboration: Some more sophisticated compilers, and many research compilers, use an analysis technique called *interprocedural analysis* to obtain more information about functions and how they are called. Interprocedural analysis attempts to discover what properties remain true across a function call. For example, we might discover that a function call can never change any global variables, which might be useful in optimizing a loop that calls such a function. Such information is called *may-information* or *flow-insensitive* information and can be obtained reasonably efficiently, although analyzing a call to a function F requires analyzing all the functions that F calls, which makes the process somewhat time consuming for large programs. A more costly property to discover is that a function *must* always change some variable; such information is called *must-information* or *flow-sensitive* information. Recall the dictate to be conservative: may-information can never be used as must-information—because a function *may* change a variable does not mean that it *must* change it. It is conservative, however, to use the negation of may-information, so the compiler can rely on the fact that a function *will* never change a variable in optimizations around the call site of that function.

One of the most important uses of interprocedural analysis is to obtain so-called alias information. An *alias* occurs when two names may designate the same variable. For example, it is quite helpful to know that two pointers passed to a function may never designate the same variable. Alias information is almost always flow-insensitive and must be used conservatively.