

Chapter 7

Optimization

7.1 Introduction and View of Optimization

In recent years, most research and development in the area of compiler design has been focused on the optimization phases of the compiler. **Optimization** is the process of improving generated code so as to reduce its potential running time and/or reduce the space required to store it in memory. Software designers are often faced with decisions which involve a space-time tradeoff – i.e., one method will result in a faster program, another method will result in a program which requires less memory, but no method will do both. However, many optimization techniques are capable of improving the object program in both time and space, which is why they are employed in most modern compilers. This results from either the fact that much effort has been directed toward the development of optimization techniques, or from the fact that the code normally generated is very poor and easily improved.

The word “optimization” is possibly a misnomer, since the techniques that have been developed simply attempt to improve the generated code, and few of them are guaranteed to produce, in any sense, optimal (the most efficient possible) code. Nevertheless, the word “optimization” is the one that is universally used to describe these techniques, and we will use it also. We have already seen that some of these techniques (such as register allocation) are normally handled in the code generation phase, and we will not discuss them here.

Optimization techniques can be separated into two general classes: local and global. **Local optimization** techniques normally are concerned with transformations on small sections of code (involving only a few instructions) and generally operate on the machine language instructions which are produced by the code generator. On the other hand, **global optimization** techniques are generally concerned with larger blocks of code, or even multiple blocks or modules, and will be applied to the intermediate form, atom

strings, or syntax trees put out by the parser. Both local and global optimization phases are optional, but may be included in the compiler as shown in Figure 7.1, i.e., the output of the parser is the input to the global optimization phase, the output of the global optimization phase is the input to the code generator, the output of the code generator is the input to the local optimization phase, and the output of the local optimization phase is the final output of the compiler. The three compiler phases shown in Figure 7.1 make up the back end of the compiler, discussed in Section 6.1.

In this discussion on improving performance, we stress the single most important property of a compiler – that it preserve the semantics of the source program. In other words, the purpose and behavior of the object program should be exactly as specified by the source program for all possible inputs. There are no conceivable improvements in efficiency which can justify violating this promise.

Having made this point, there are frequently situations in which the computation specified by the source program is ambiguous or unclear for a particular computer architecture. For example, in the expression $(a + b) * (c + d)$ the compiler will have to decide which addition is to be performed first (assuming that the target machine has only one Arithmetic and Logic Unit). Most programming languages leave this unspecified, and it is entirely up to the compiler designer, so that different compilers could evaluate this expression in different ways. In most cases it may not matter, but if any of a , b , c , or d happen to be function calls which produce output or side effects, it may make a significant difference. Languages such as C, Lisp, and APL, which have assignment operators, yield an even more interesting example:

`a = 2; b = (a * 1 + (a = 3));`

Some compiler writers feel that programmers who use ambiguous expressions such as these deserve whatever the compiler may do to them.

A fundamental question of philosophy is inevitable in the design of the optimization phases. Should the compiler make extensive transformations and improvements to the source program, or should it respect the programmer's decision to do things that are inefficient or unnecessary? Most compilers tend to assume that the average programmer does not intentionally write inefficient code, and will perform the optimizing transformations. A sophisticated programmer or hacker who, in rare cases, has a reason for writing the code in that fashion can usually find a way to force the compiler to generate the desired output.

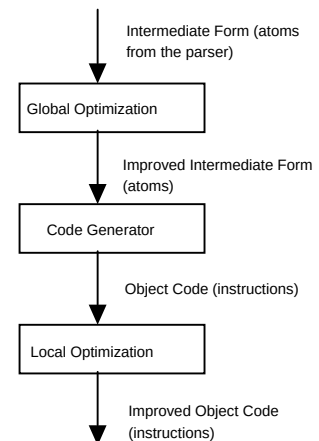


Figure 7.1 Sequence of Optimization Phases in a Compiler

One significant problem for the user of the compiler, introduced by the optimization phases, has to do with debugging. Many of the optimization techniques will remove unnecessary code and move code within the object program to an extent that run-time debugging is affected. The programmer may attempt to step through a series of statements which either don't exist, or occur in an order different from what was originally specified by the source program!

To solve this problem, most modern and available compilers include a switch with which optimization may be turned on or off. When debugging new software, the switch is off, and when the software is fully tested, the switch can be turned on to produce an efficient version of the program for distribution. It is essential, however, that the optimized version and the non-optimized version be functionally equivalent (i.e., given the same inputs, they should produce identical outputs). This is one of the more difficult problems that the compiler designer must deal with.

Another solution to this problem, used by IBM in the early 1970's for its PL/1 compiler, is to produce two separate compilers. The **checkout compiler** was designed for interactive use and debugging. The **optimizing compiler** contained extensive optimization, but was not amenable to the testing and development of software. Again, the vendor (IBM in this case) had to be certain that the two compilers produced functionally equivalent output.

Exercises 7.1

1. Using a C++ compiler,
 - (a) what would be printed as a result of running the following:

```
{
int a;
(a = 2) + (a = 3);
cout << a;
}
```

- (b) What other value might be printed as a result of compilation with a different compiler?

2. Explain why the following two statements cannot be assumed to be equivalent:

$$a = f(x) + f(x) + f(x) ;$$

$$a = 3 * f(x) ;$$

3. (a) Perform the following computations, rounding to four significant digits after each operation.

$$(0.7043 + 0.4045) + -0.3330 = ?$$

$$0.7043 + (0.4045 + -0.3330) = ?$$

- (b) What can you conclude about the associativity of addition with computer arithmetic?

7.2 Global Optimization

As mentioned previously, **global optimization** is a transformation on the output of the parser. Global optimization techniques will normally accept, as input, the intermediate form as a sequence of atoms (three-address code) or syntax trees. There are several global optimization techniques in the literature – more than we can hope to cover in detail. Therefore, we will look at the optimization of common subexpressions in basic blocks in some detail, and then briefly survey some of the other global optimization techniques.

A few optimization techniques, such as algebraic optimizations, can be considered either local or global. Since it is generally easier to deal with atoms than with instructions, we will include algebraic techniques in this section.

7.2.1 Basic Blocks and DAGs

The sequence of atoms put out by the parser is clearly not an optimal sequence; there are many unnecessary and redundant atoms. For example, consider the C++ statement:

```
a = (b + c) * (b + c) ;
```

The sequence of atoms put out by the parser could conceivably be as shown in Figure 7.2 below:

```
(ADD, b, c, T1)
(ADD, b, c, T2)
(MUL, T1, T2, T3)
(MOV, T3, , a)
```

Figure 7.2 Atom Sequence for `a = (b + c) * (b + c) ;`

Every time the parser finds a correctly formed addition operation with two operands it blindly puts out an ADD atom, whether or not this is necessary. In the above example, it is clearly not necessary to evaluate the sum `b + c` twice. In addition, the MOV atom is not necessary because the MUL atom could store its result directly into the variable `a`. The atom sequence shown in Figure 7.3, below, is equivalent to the one given in Figure 7.2, but requires only two atoms because it makes use of common subexpressions and it stores the result in the variable `a`, rather than a temporary location.

```
(ADD, b, c, T1)
(MUL, T1, T1, a)
```

Figure 7.3 Optimized Atom Sequence for `a = (b + c) * (b + c) ;`

In this section, we will demonstrate some techniques for implementing these optimization improvements to the atoms put out by the parser. These improvements will result in programs which are both smaller and faster, i.e., they optimize in both space and time.

It is important to recognize that these optimizations would not have been possible if there had been intervening Label or Jump atoms in the parser output. For example, if the atom sequence had been as shown in Figure 7.4, we could not have optimized to the sequence of Figure 7.3, because there could be atoms which jump into this code at Label L1, thus altering our assumptions about the values of the variables and temporary locations. (The atoms in Figure 7.4 do not result from the given C++ state-

```
(ADD, b, c, T1)
(LBL, L1)
(ADD, b, c, T2)
(MUL, T1, T2, T3)
(TST, b, c,, 1, L3)
(MOV, T3,, a)
```

Figure 7.4 Example of an Atom Sequence Which Cannot be Optimized

ment, and the example is, admittedly, artificially contrived to make the point that Label atoms will affect our ability to optimize.)

By the same reasoning, Jump or Branch atoms will interfere with our ability to make these optimizing transformations to the atom sequence. In Figure 7.4 the MUL atom cannot store its result into the variable `a`, because the compiler does not know whether the conditional branch will be taken.

The optimization techniques which we will demonstrate can be effected only in certain subsequences of the atom string, which we call **basic blocks**. A basic block is a section of atoms which contains no Label or branch atoms (i.e., LBL, TST, JMP). In Figure 7.5, we show that the atom sequence of Figure 7.4 is divided into three basic blocks.

Each basic block is optimized as a separate entity. There are more advanced techniques which permit optimization across basic blocks, but they are beyond the scope of this text. We use a **Directed Acyclic Graph**, or **DAG**, to implement this optimization. The DAG is *directed* because the arcs have arrows indicating the direction of the arcs, and it is *acyclic* because there is no path leading from a node back to itself (i.e., it has no

(ADD, b, c, T1)	Block 1
(LBL, L1)	
(ADD, b, c, T2)	Block 2
(MUL, T1, T2, T3)	
(TST, b, c,, 1, L3)	
(MOV, T3,, a)	Block 3

Figure 7.5 Basic Blocks Contain No LBL, TST, or JMP Atoms

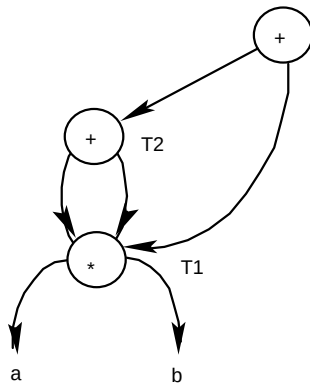


Figure 7.6 Example of a DAG

cycles). The DAG is similar to a syntax tree, but it is not truly a tree because some nodes may have more than one parent and also because the children of a node need not be distinct. An example of a DAG, in which interior nodes are labeled with operations, and leaf nodes are labeled with operands, is shown, below, in Figure 7.6.

Each of the operations in Figure 7.6 is a binary operation (i.e., each operation has two operands), consequently each interior node has two arcs pointing to the two operands. Note that in general we will distinguish between the left and right arc because we need to distinguish between the left and right operands of an operation

(this is certainly true for subtraction and division, which are not commutative operations). We will be careful to draw the DAGs so that it is always clear which arc represents the left operand and which arc represents the right operand. For example, in Figure 7.6 the left operand of the addition labeled T3 is T2, and the right operand is T1. Our plan is to show how to build a DAG from an atom sequence, from which we can then optimize the atom sequence.

We will begin by building DAGs for simple arithmetic expressions. DAGs can also be used to optimize complete assignment statements and blocks of statements, but we will not take the time to do that here. To build a DAG, given a sequence of atoms representing an arithmetic expression with binary operations, we use the following algorithm:

1. Read an atom.
2. If the operation and operands match part of the existing DAG (i.e., if they form a sub DAG), then add the result Label to the list of Labels on the parent and repeat from Step 1. Otherwise, allocate a new node for each operand that is not already in the DAG, and a node for the operation. Label the operation node with the name of the result of the operation.
3. Connect the operation node to the two operands with directed arcs, so that it is clear which operand is the left and which is the right.
4. Repeat from Step 1.

As an example, we will build a DAG for the expression $a * b + a * b + a * b$. This expression clearly has some common subexpressions, which should make it amenable for optimization. The atom sequence as put out by the parser would be:

```
(MUL, a, b, T1)
(MUL, a, b, T2)
(ADD, T1, T2, T3)
```

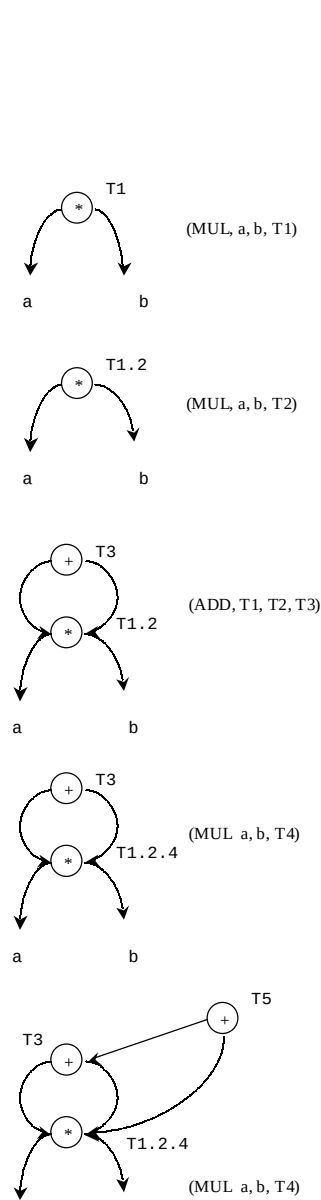


Figure 7.7 Building the DAG for $a * b + a * b + a * b$

(MUL, a, b, T4)
(ADD, T3, T4, T5)

We follow the algorithm to build the DAG, as shown in Figure 7.7, in which we show how the DAG is constructed as each atom is processed.

The DAG is a graphical representation of the computation needed to evaluate the original expression in which we have identified common subexpressions. For example, the expression $a * b$ occurs three times in the original expression $a * b + a * b + a * b$. The three atoms corresponding to these subexpressions store results into $T1$, $T2$, and $T4$. Since the computation need be done only once, these three atoms are combined into one node in the DAG labeled $T1.2.4$. After that point, any atom which uses $T1$, $T2$, or $T4$ as an operand will point to $T1.2.4$.

We are now ready to convert the DAG to a basic block of atoms. The algorithm given below will generate atoms (in reverse order) in which all common subexpressions are evaluated only once:

1. Choose any node having no incoming arcs (initially there should be only one such node, representing the value of the entire expression).
2. Put out an atom for its operation and its operands.
3. Delete this node and its outgoing arcs from the DAG.
4. Repeat from Step 1 as long as there are still operation nodes remaining in the DAG.

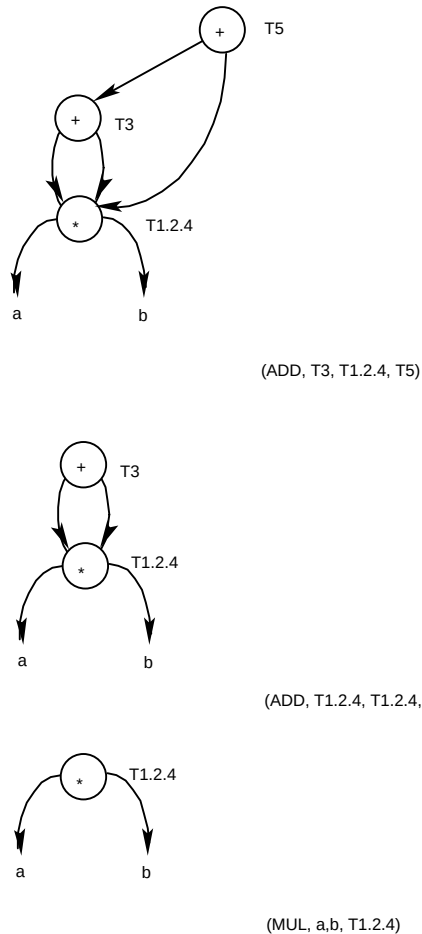


Figure 7.8 Generating Atoms from the DAG for $a * b + a * b + a * b$

This algorithm is demonstrated below, in Figure 7.8, in which we are working with the same expression that generated the DAG of Figure 7.7. The DAG and the output are shown for each iteration of the algorithm (there are three iterations).

A composite node, such as $T1.2.4$, is referred to by its full name rather than simply $T1$ or $T2$ by convention, and to help check for mistakes. The student should verify that the three atoms generated in Figure 7.8 actually compute the given expression, reading the atoms from bottom to top. We started with a string of five atoms, and have improved it to an equivalent string of only three atoms. This will result in significant savings in both run time and space required for the object program.

Unary operations can be handled easily using this method. Since a unary operation has only one operand, its node will have only one arc pointing to the operand, but in all other respects the algorithms given for building DAGs and generating optimized atom sequences remain unchanged. Consequently, this method generalizes well to expressions involving operations with any number of operands, though for our purposes operations will generally have two operands.

Sample Problem 7.2 (a)

Construct the DAG and show the optimized sequence of atoms for the C++ expression $(a - b) * c + d * (a - b) * c$. The atoms produced by the parser are shown below:

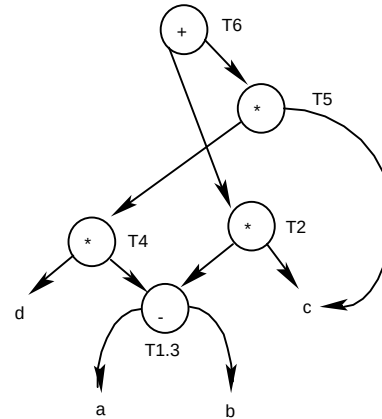
```
(SUB, a, b, T1)
(MUL, T1, c, T2)
(SUB, a, b, T3)
(MUL, d, T3, T4)
(MUL, T4, c, T5)
(ADD, T2, T5, T6)
```

Solution:

```

(SUB, a, b, T1.3)
(MUL, d, T1.3, T4)
(MUL, T4, c, T5)
(MUL, T1.3, c, T2)
(ADD, T2, T5, T6)

```

**7.2.2 Other Global Optimization Techniques**

We will now examine a few other common global optimization techniques, however, we will not go into the implementation of these techniques.

Unreachable code is an atom or sequence of atoms which cannot be executed because there is no way for the flow of control to reach that sequence of atoms. For example, in the following atom sequence the MUL, SUB, and ADD atoms will never be executed because of the unconditional jump preceding them.

(JMP, L1)		
(MUL, a, b, T1)		
(SUB, T1, c, T2)	⇒	(JMP, L1)
(ADD, T2, d, T3)		(LBL, L2)
(LBL, L2)		

Thus, the three atoms following the JMP and preceding the LBL can all be removed from the program without changing the purpose of the program. In general, a JMP atom should always be followed by an LBL atom. If this is not the case, simply remove the intervening atoms between the JMP and the next LBL.

Data flow analysis is a formal way of tracing the way information about data items moves through the program and is used for many optimization techniques. Though data flow analysis is beyond the scope of this text, we will look at some of the optimizations that can result from this kind of analysis.

One such optimization technique is **elimination of dead code**, which involves determining whether computations specified in the source program are actually used and affect the program's output. For example, the program in Figure 7.9 contains an assignment to the variable *a* which has no effect on the output since *a* is not used subsequently, but prior to another assignment to the variable *a*.

```

{
    a = b + c * d;    //This statement has no effect and can be removed
    b = c * d / e;
    c = b - 3;
    a = b - c;
    cout << a << b << c ;
}

```

Figure 7.9 Elimination of Dead Code

Another optimization technique which makes use of data flow analysis is the detection of loop invariants. A **loop invariant** is code within a loop which deals with data values that remain constant as the loop repeats. Such code can be moved outside the loop, causing improved run time without changing the program's semantics. An example of loop invariant code is the call to the square root function (`sqrt`) in the program of Figure 7.10, below.

Since the value assigned to `a` is the same each time the loop repeats, there is no need for it to be repeated; it can be done once before entering the loop (we need to be sure, however, that the loop is certain to be executed at least once). This optimization will eliminate 999 unnecessary calls to the `sqrt` function.

The remaining global optimization techniques to be examined in this section all involve mathematical transformations. The student is cautioned that their use is not universally recommended, and that it is often possible, by employing them, that the compiler designer is effecting transformations which are undesirable to the source programmer. For example, the question of the meaning of **arithmetic overflow** is crucial here. If the unoptimized program reaches an overflow condition for a particular input, is it valid for the optimized program to avoid the overflow? (Be careful; most computers have run-time traps designed to transfer control to handle conditions such as overflow. It

```

{
    for (i=0; i<1000; i++)
    { a = sqrt (x);          // loop invariant
      vector[i] = i * a;
    }
}

{
    a = sqrt (x);          // loop invariant
    for (i=0; i<1000; i++)
    {
        vector[i] = i * a;
    }
}

```

Figure 7.10 Movement of Loop Invariant Code

```

{
    a = 2 * 3;           // a must be 6
    b = c + a * a;       // a * a must be 36
}

{
    a = 6;
    b = c + 36;
}

```

Figure 7.11 Constant Folding

could be that the programmer intended to trap certain input conditions.) There is no right or wrong answer to this question, but it is an important consideration when implementing optimization.

Constant folding is the process of detecting operations on constants, which could be done at compile time rather than run time. An example is shown, above, in Figure 7.11 in which the value of the variable *a* is known to be 6, and the value of the expression *a * a* is known to be 36. If these computations occur in a small loop, constant folding can result in significant improvement in run time (at the expense of a little compile time).

Another mathematical transformation is called **reduction in strength**. This optimization results from the fact that certain operations require more time than others on virtually all architectures. For example, multiplication can be expected to be significantly more time consuming than addition. Thus, the multiplication $2 * x$ is certain to be slower than the addition $x + x$. Likewise, if there is an exponentiation operator, x^2 is certain to be slower than $x * x$.

A similar use of reduction in strength involves using the shift instructions available on most architectures to speed up fixed point multiplication and division. A multiplication by a positive power of two is equivalent to a left shift, and a division by a positive power of two is equivalent to a right shift. For example, the multiplication $x * 8$ can be done faster simply by shifting the value of *x* three bit positions to the left, and the division $x / 32$ can be done faster by shifting the value of *x* five bit positions to the right.

Our final example of mathematical transformations involves **algebraic transformations** using properties such as commutativity, associativity, and the distributive property, all summarized, below, in Figure 7.12. We do not believe that these properties

$a + b == b + a$	Addition is commutative
$(a + b) + c == a + (b + c)$	Addition is associative
$a * (b + c) == a * b + a * c$	Multiplication distributes over addition

Figure 7.12 Algebraic Identities

are necessarily true when dealing with computer arithmetic, due to the finite precision of numeric data. Nevertheless, they are employed in many compilers, so we give a brief discussion of them here.

Though these properties are certainly true in mathematics, they do not necessarily hold in computer arithmetic, which has finite precision and is subject to overflow in both fixed-point and floating-point representations. Thus, the decision to make use of these properties must take into consideration the programs which will behave differently with optimization put into effect. At the very least, a warning to the user is recommended for the compiler's user manual.

The discussion of common subexpressions in Section 7.2.1 would not have recognized any common subexpressions in the following:

$a = b + c;$

$b = c + d + b;$

but by employing the commutative property, we can eliminate an unnecessary computation of $b + c$:

$a = b + c;$

$b = a + d;$

A multiplication operation can be eliminated from the expression $a * c + b * c$ by using the distributive property to obtain $(a + b) * c$.

Compiler writers who employ these techniques create more efficient programs for the large number of programmers who want and appreciate the improvements, but risk generating unwanted code for the small number of programmers who require that algebraic expressions be evaluated exactly as specified in the source program.

Sample Problem 7.2 (b)

Use the methods of unreachable code, constant folding, reduction in strength, loop invariants, and dead code to optimize the following atom stream; you may assume that the TST condition is initially not satisfied:

```
(LBL, L1)
(TST, a, b,, 1, L2)
(SUB, a, 1, a)
(MUL, x, 2, b)
(ADD, x, y, z)
(ADD, 2, 3, z)
(JMP, L1)
(SUB, a, b, a)
(MUL, x, 2, z)
(LBL, L2)
```

Solution:

(LBL, L1)	
(TST, a, b,, 1, L2)	
(SUB, a, 1, a)	
(MUL, x, 2, b)	Reduction in strength
(ADD, x, y, z)	Elimination of dead code
(ADD, 2, 3, z)	Constant folding, loop invariant
(JMP, L1)	
(SUB, a, b, a)	Unreachable code
(MUL, x, 2, z)	Unreachable code
(LBL, L2)	
(MOV, 5,, z)	
(LBL, L1)	
(TST, a, b,, 1, L2)	
(SUB, a, 1, a)	
(ADD, x, x, b)	
(JMP, L1)	
(LBL, L2)	

Exercises 7.2

1. Eliminate *common subexpressions* from each of the following strings of atoms, using DAGs as shown in Sample Problem 7.2 (a) (we also give the C++ expressions from which the atom strings were generated):

(a) $(b + c) * d * (b + c)$

```
(ADD, b, c, T1)
(MUL, T1, d, T2)
(ADD, b, c, T3)
(MUL, T2, T3, T4)
```

(b) $(a + b) * c / ((a + b) * c - d)$

```
(ADD, a, b, T1)
(MUL, T1, c, T2)
(ADD, a, b, T3)
(MUL, T3, c, T4)
```

```
(SUB, T4, d, T5)
(DIV, T2, T5, T6)
```

(c) $(a + b) * (a + b) - (a + b) * (a + b)$

```
(ADD, a, b, T1)
(ADD, a, b, T2)
(MUL, T1, T2, T3)
(ADD, a, b, T4)
(ADD, a, b, T5)
(MUL, T4, T5, T6)
(SUB, T3, T6, T7)
```

(d) $((a + b) + c) / (a + b + c) - (a + b + c)$

```
(ADD, a, b, T1)
(ADD, T1, c, T2)
(ADD, a, b, T3)
(ADD, T3, c, T4)
(DIV, T2, T4, T5)
(ADD, a, b, T6)
(ADD, T6, c, T7)
(SUB, T5, T7, T8)
```

(e) $a / b - c / d - e / f$

```
(DIV, a, b, T1)
(DIV, c, d, T2)
(SUB, T1, T2, T3)
(DIV, e, f, T4)
(SUB, T3, T4, T5)
```

2. How many different *atom sequences* can be generated from the DAG given in your response to Problem 1 (e), above?

3. In each of the following sequences of atoms, eliminate the *unreachable atoms*:

(a) (ADD, a, b, T1)
 (LBL, L1)
 (SUB, b, a, b)
 (TST, a, b, , 1, L1)
 (ADD, a, b, T3)
 (JMP, L1)

(b) (ADD, a, b, T1)
 (LBL, L1)
 (SUB, b, a, b)
 (JMP, L1)
 (ADD, a, b, T3)
 (LBL, L2)

(c) (JMP, L2)
 (ADD, a, b, T1)
 (TST, a, b, , 3, L2)
 (SUB, b, b, T3)
 (LBL, L2)
 (MUL, a, b, T4)

4. In each of the following C++ functions, eliminate statements which constitute *dead code*. In each case, the procedure returns a value to the calling procedure in the parameter d:

(a)

```
void f (int & d)
{ int a,b,c;
  a = 3;
  b = 4;
  d = a * b + d;
}
```

(b)

```
void f (int & d)
{ int a,b,c;
  a = 3;
  b = 4;
```

```

        c = a + b;
        d = a + b;
        a = b + c * d;
        b = a + c;
    }

```

5. In each of the following C++ program segments, optimize the loops by moving *loop invariant code* outside the loop:

```

(a)    {    for (i=0; i<100; i++)
           {        a = x[i] + 2 * a;
                   b = x[i];
                   c = sqrt (100 * c);
           }
    }

```

```

(b)    {    for (j=0; j<50; j++)
           {        a = sqrt (x);
                   n = n * 2;
                   for (i=0; i<10; i++)
                       {        y = x;
                               b[n] = 0;
                               b[i] = 0;
                       }
           }
    }

```

6. Show how *constant folding* can be used to optimize the following C++ program segments:

```

(a)    a = 2 + 3 * 8;
        b = b + (a - 3);

```

```
(b)  function f (int & c)
      {      const int a = 44;
              const int b = a - 12;
              c = a + b - 7;
      }
```

7. Use *reduction in strength* to optimize the following sequences of atoms. Assume that there are (SHL, x, y, z) and (SHR, x, y, z) atoms which will shift x left or right respectively by y bit positions, leaving the result in z (also assume that these are fixed-point operations):

```
(a)  (MUL, x, 2, T1)
      (MUL, y, 2, T2)
```

```
(b)  (MUL, x, 8, T1)
      (DIV, y, 16, T2)
```

8. Which of the following optimization techniques, when applied successfully, will always result in *improved execution time*? Which will result in *reduced program size*?

- (a) Detection of common subexpressions with DAGs
- (b) Elimination of unreachable code
- (c) Elimination of dead code
- (d) Movement of loop invariants outside of loop
- (e) Constant folding
- (f) Reduction in strength

7.3 Local Optimization

In this section we discuss **local optimization** techniques. The definition of *local* versus *global* techniques varies considerably among compiler design textbooks. Our view is that any optimization which is applied to the generated code is considered local. Local optimization techniques are often called *peephole* optimization, since they generally involve transformations on instructions which are close together in the object program. The student can visualize them as if peering through a small peephole at the generated code.

There are three types of local optimization techniques which will be discussed here: load/store optimization, jump over jump optimization, and simple algebraic optimization. In addition, register allocation schemes such as the one discussed in Section 6.4 could be considered local optimization, though they are generally handled in the code generator itself.

The parser would translate the expression $a + b - c$ into the following stream of atoms:

```
(ADD, a, b, T1)
(SUB, T1, c, T2)
```

The simplest code generator design, as presented in Chapter 6, would generate three instructions corresponding to each atom: Load the first operand into a register (LOD), perform the operation, and store the result back to memory (STO). The code generator would then produce the following instructions from the atoms:

```
LOD    R1, a
ADD    R1, b
STO    R1, T1
LOD    R1, T1
SUB    R1, c
STO    R1, T2
```

Notice that the third and fourth instructions in this sequence are entirely unnecessary since the value being stored and loaded is already at its destination. The above sequence of six instructions can be optimized to the following sequence of four instructions by eliminating the intermediate Load and Store instructions as shown below:

```
LOD    R1, a
ADD    R1, b
SUB    R1, c
STO    R1, T2
```

For lack of a better term, we call this a **load/store optimization**. It is clearly machine dependent.

Another local optimization technique, which we call a **jump over jump optimization**, is very common and has to do with unnecessary jumps. The student has already seen examples in Chapter 4 of conditional jumps in which it is clear that greater efficiency can be obtained by rewriting the conditional logic. A good example of this can be found in a C++ compiler for the statement `if (a>b) a = b;`. It might be translated into the following stream of atoms:

```
(TST, a, b, , 3, L1)
(JMP, L2)
(LBL, L1)
(MOV, b, , a)
(LBL, L2)
```

A reading of this atom stream is “Test for a greater than b, and if true, jump to the assignment. Otherwise, jump around the assignment.” The reason for this somewhat convoluted logic is that the TST atom uses the same comparison code found in the expression. The instructions generated by the code generator from this atom stream would be:

```

        LOD    R1, a
        CMP    R1, b, 3           //Is R1 > b?
        JMP    L1
        CMP    0, 0, 0           // Unconditional Jump
        JMP    L2
L1:
        LOD    R1, b
        STO    R1, a
L2:
```

It is not necessary to implement this logic with two Jump instructions. We can improve this code significantly by testing for the condition to be false rather than true, as shown below:

```

        LOD    R1, a
        CMP    R1, b, 4           // Is R1 - b?
        JMP    L1
        LOD    R1, b
        STO    R1, a
L1:
```

This optimization could have occurred in the intermediate form (i.e., we could have considered it a global optimization), but this kind of jump over jump can occur for various other reasons. For example, in some architectures, a conditional jump is a “short” jump (to a restricted range of addresses), and an unconditional jump is a “long” jump. Thus, it is not known until code has been generated whether the target of a

conditional jump is within reach, or whether an unconditional jump is needed to jump that far.

The final example of local optimization techniques involves simple algebraic transformations which are machine dependent and are called *simple algebraic optimizations*. For example, the following instructions can be eliminated:

```
MUL    R1, 1
ADD    R1, 0
```

because multiplying a value by 1, or adding 0 to a value, should not change that value. (Be sure, though, that the instruction has not been inserted to alter the condition code or flags register.) In addition, the instruction (MUL R1, 0) can be improved by replacing it with (CLR R1), because the result will always be 0 (this is actually a reduction in strength transformation).

Sample Problem 7.3

Use the peephole methods of load/store, jump over jump, and simple algebraic optimization to improve the following Mini program segment:

```

        CMP    R1, a, 2                // JMP if R1 < a
        JMP    L1
        CMP    0, 0, 0
        JMP    L2
L1:
        LOD    R1, b
        ADD    R1, c
        STO    R1, T1
        LOD    R1, T1
        SUB    R1, a
        STO    R1, T2
        LOD    R1, T2
        STO    R1, a
        SUB    R1, 0
        STO    R1, b
L2:
```

Solution:

```

        CMP    R1, a, 2                // Jump Over Jump
        JMP    L1
        CMP    0, 0, 0
        JMP    L2
```

```

L1:
    LOD    R1, b
    ADD    R1, c
    ST0    R1, T1           // Load/Store
    LOD    R1, T1
    SUB    R1, a
    ST0    R1, T2           // Load/Store
    LOD    R1, T2
    ST0    R1, a
    SUB    R1, 0            // Algebraic
    ST0    R1, b

L2:

    // optimized code
    CMP    R1, a, 5         // JMP if R1 ≥ a
    JMP    L2
    LOD    R1, b
    ADD    R1, c
    SUB    R1, a
    ST0    R1, a
    ST0    R1, b

L2:

```

Exercises 7.3

1. Optimize each of the following code segments for unnecessary *Load/Store* instructions:

(a)

```

LOD    R1, a
ADD    R1, b
ST0    R1, T1
LOD    R1, T1
SUB    R1, c
ST0    R1, T2
LOD    R1, T2
ST0    R1, d

```

(b)

```

LOD    R1, a
LOD    R2, c
ADD    R1, b
ADD    R2, b
ST0    R2, T1
ADD    R1, c
LOD    R2, T1
ST0    R1, T2
ST0    R2, c

```

2. Optimize each of the following code segments for unnecessary *jump over jump* instructions:

(a) <pre> CMP R1, a, 1 JMP L1 CMP 0, 0, 0 JMP L2 L1: ADD R1, R2 L2: </pre>	(b) <pre> CMP R1, a, 5 JMP L1 CMP 0, 0, 0 JMP L2 L1: SUB R1, a L2: </pre>
(c) <pre> L1: ADD R1, R2 CMP R1, R2, 3 JMP L2 CMP 0, 0, 0 JMP L1 L2: </pre>	

3. Use any of the *local optimization* methods of this section to optimize the following code segment:

```

CMP    R1, R2, 6                // JMP if R1 ≠ R2
JMP    L1
CMP    0, 0, 0
JMP    L2

L1:
LOD    R2, a
ADD    R2, b
STO    R2, T1
LOD    R2, T1
MUL    R2, c
STO    R2, T2
LOD    R2, T2
STO    R2, d
SUB    R1, 0
STO    R1, b

L2:

```

7.4 Chapter Summary

Optimization has to do with the improvement of machine code and/or intermediate code generated by other phases of the compiler. These improvements can result in reduced run time and/or space for the object program. There are two main classifications of optimization: global and local. *Global optimization* operates on atoms or syntax trees put out by the front end of the compiler, and *local optimization* operates on instructions put out by the code generator. The term “optimization” is used for this phase of the compiler, even though it is never certain to produce optimal code in either space or time.

The compiler writer must be careful not to change the intent of the program when applying optimizing techniques. Many of these techniques can have a profound effect on debugging tools; consequently, debugging is generally done on unoptimized code.

Global optimization is applied to blocks of code in the intermediate form (atoms) which contain no Label or branch atoms. These are called *basic blocks*, and they can be represented by *directed acyclic graphs* (DAGs), in which each interior node represents an operation with links to its operands. We show how the DAGs can be used to optimize common subexpressions in an arithmetic expression.

We briefly describe a few more global optimization techniques without going into the details of their implementation. They include: (1) *unreachable code* – code which can never be executed and can therefore be eliminated; (2) *dead code* – code which may be executed but can not have any effect on the program's output and can therefore be eliminated; (3) *loop invariant code* – code which is inside a loop, but which doesn't really need to be in the loop and can be moved out of the loop; (4) *constant folding* – detecting arithmetic operations on constants which can be computed at compile time rather than at run time; (5) *reduction in strength* – substituting a faster arithmetic operation for a slow one; (6) *algebraic transformations* – transformations involving the commutative, associative, and distributive properties of arithmetic.

We describe three types of *local optimization*: (1) *load/store optimization* – eliminating unnecessary Load and Store instructions in a Load/Store architecture; (2) *jump over jump optimizations* – replacing two Jump instructions with a single Jump by inverting the logic; (3) *simple algebraic optimization* – eliminating an addition or subtraction of 0 or a multiplication or division by 1.

These optimization techniques are optional, but they are used in most modern compilers because of the resultant improvements to the object program, which are significant.