Chapter 6

Optimization

- Code as generated by syntax-directed translation tends to contain a lot of deadwood.
  - There are redundant instructions
  - And operations are implemented in clumsy ways.
- Optimization is the attempt to reach the ideal.
  - The type of code produced by an expert assembly-language programmer on a familiar machine.

- The optimizer's problem is to remove as much of the deadwood as possible without changing the meaning of the program.
- But, There are limits to what optimization can do. Indeed, optimization may on rare occasions make a program bigger or slower than it was before.
- Moreover, optimization is no substitute for good program design, or especially for intelligent algorithm selection.

- Most optimization consists of identifying unnecessary computations.
  - We have seen that DAGs can be used for this purpose in short sequences of instructions.
  - For larger-scale analysis, we must have a way of tracking the way information about data items moves through the program;
    - This is known as *data-flow analysis*, and it is one of our main tools in optimization.

1. Machine Independent Optimization

- We classify optimization techniques as machine dependent or machine independent.
  - Machine dependent optimization consists of finding convenient shortcuts that take advantage of the architecture and instruction set of the target machine.
  - Machine independent optimization analyzes the organization of the program itself at the intermediate-code level. This is more general, so we will begin with it.

1.1 Basic Blocks

- Some optimization can be done within individual statements, but the most fruitful source of deadwood is in redundancies over sequences of statements.
- For this reason, most machine-independent optimization takes as its starting point the *basic block*.

- A basic block is a set of 3AC instructions that are always executed sequentially from beginning to end.
  - This means that there are no branches into or out of the middle of the block
  - and that the entire program can be represented as a set of basic blocks interconnected by branches.
    - The branches indeed define the boundaries of the blocks.

- A basic block begins
  - At the start of the program,
  - At the start of a procedure
  - At the target of any branch
  - Immediately after a branch
- And it ends
  - Before the next basic block
  - At the end of the program
  - At the end of a procedure.
- It is easiest to identify the starts of the blocks; then each block continues until the start of the next one or the end of the program.

- We can now distinguish three levels of optimization:
  - local optimization
    - which limits its scope to the current basic block.
  - global optimization
    - which must take into account the flow of information between blocks.
  - interprocedural optimization
    - which considers the flow of information between procedures.
- We can generally do local with a DAG, the others require data-flow analysis (Sec 6.3)

■ First, let's break a procedure into basic blocks.

```
1   procedure SORT (n: integer; var x: sortarray);
2      var
3         i, jmax, j, temp: integer;
4      begin
5      for i := n downto 2 do
6         begin
7         jmax := 1;
8         for j := 2 to i do          { Find largest element  }
9            if x[j] > x[jmax] then
10              jmax := j;
11        if jmax <> i then
12           begin                    { Swap into place       }
13           temp := x[i];
14           x[i] := x[jmax];
15           x[jmax] := temp;
16           end
17        end
18     end;  { Sort }
```


■ 3AC

```
 (1)  i := n                         Line 5
 (2)  if i < 2 then goto (32)
 (3)  jmax := 1                      Line 7
 (4)  j := 2                         Line 8
 (5)  if j > i then goto (16)
 (6)  T1 := j - 1                    Line 9
 (7)  T2 := T1*4
 (8)  T3 := x[T2]                    T3 holds x[j]
 (9)  T4 := jmax - 1
(10)  T5 := T4*4
(11)  T6 := x[T5]                    T6 holds x[jmax]
(12)  if T3 <= T6 then goto (14)
(13)  jmax := j                      Line 10
(14)  j := j + 1
(15)  goto (5)
(16)  if jmax = i then goto (30)     Line 11
(17)  T7 := i - 1                    Line 13
(18)  T8 := T7*4
(19)  T9 := x[T8]                    T9 holds x[i]
(20)  temp := T9                     temp := x[i]
(21)  T10 := jmax - 1
(22)  T11 := T10*4
(23)  T12 := x[T11]                  T12 holds x[jmax]
(24)  T13 := i - 1                   Line 14
(25)  T14 := T13*4
(26)  x[T14] := T12                  x[i] := x[jmax]
(27)  T15 := jmax - 1                Line 15
(28)  T16 := T15*4
(29)  x[T16] := temp                 x[jmax] := temp
(30)  i := i - 1
(31)  goto (2)
(32)  return
```


■ Basic Blocks.

(1) i := n	B_1 (Start of procedure)
(2) if i < 2 then goto B_{11}	B_2 (Target of goto, Line 31)
(3) jmax := 1 (4) j := 2	B_3 (Follows conditional branch)
(5) if j > i then goto B_8	B_4 (Target of goto, Line 15)
(6) T1 := j - 1 (7) T2 := T1*4 (8) T3 := x[T2] (9) T4 := jmax - 1 (10) T5 := T4*4 (11) T6 := x[T5] (12) if T3 <= T6 then goto B_7	B_5 (Follows conditional branch)
(13) jmax := j	B_6 (Follows conditional branch)
(14) j := j + 1 (15) goto B_4	B_7 (Target of goto, Line 12)
(16) if i = jmax then goto B_{10}	B_8 (Target of goto, Line 5)
(17) T7 := i - 1 (18) T8 := T7*4 (19) T9 := x[T8] (20) temp := T9 (21) T10 := jmax - 1 (22) T11 := T10*4 (23) T12 := x[T11] (24) T13 := i - 1 (25) T14 := T13*4 (26) x[T14] := T12 (27) T15 := jmax - 1 (28) T16 := T15*4 (29) x[T16] := temp	B_9 (Follows conditional branch)
(30) i := i - 1 (31) goto B_2	B_{10} (Target of goto, Line 16)
(32) return	B_{11} (Target of goto, Line 2)

- In the course of the ensuing optimizations, instructions within a basic block may be modified or deleted.
  - Hence it is a good idea to make the gotos refer to block numbers instead of quadruple numbers, in case the target quad gets deleted.
  - You may have noticed that this was done in the breakdown of the previous basic blocks.

- At the moment we are interested in the individual blocks; but when we come to global optimization, we will also be concerned with the flow of control and of data between the blocks.
- The gotos in the code can be pictured as creating paths between the blocks; when we show these paths, we get a flow graph.
- The flow graph for the basic blocks we just created is shown on the next slide.

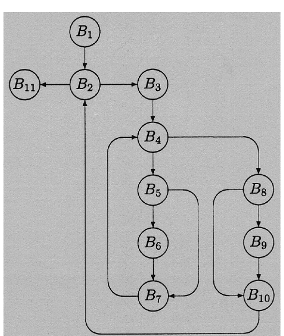


Figure 6.1

1.2 Local Optimization

- We can do a significant variety of enhancements within a basic block.
- Some of these optimizations can be done globally as well as locally; I am listing them here because then *can* be done locally.
- Details about their implementation must await the development of theory later in the chapter.

- Constant Folding.
  - If a program contains a subexpression whose operands are constants, like
    - x := 2 * 3;
  - There is no need to multiply these numbers at run time; the compiler can compute the product and substitute
    - x := 6;

- Programmers rarely write anything like that; but good programmers will frequently write
  - const
    - lgth = 2;
    - amount = 3;
  - ...
  - x:= lgth * amount;
- Programming practices give rise to operations on constants.
- Precomputing the results of such operations at compile time is called *constant folding* and is one of the most common types of optimization.

- The picture is more complicated if the programmer has scattered constants about the statement, as, for example, in
  - x := lgth*(b + c/a)*amount;
- From axioms in basic math you would say you could re-order the statement to be
  - x := lgth*amount*(b+c/a);
  - Don't even think about this if anything in the statement is not an integer, because the order change can change the final result.

- Another, less obvious, opportunity occurs when a programmer assigns a value to a variable once and never changes the value again.
  - var.
    - lgth : integer;
  - lgth:=2;
  - For all practical purposes lgth is a constant, but using it throughout the program in an optimization technique is called *constant propagation*.
  - If a computed value, like lgth, is to be propagated through the entire program, then we need to do data-flow analysis to guarantee that the value is the same regardless of the path taken through the program at run time.

- Copy Propagation.
  - Copy instructions are 3AC instructions of the form
    - x := y;
  - When instructions of this sort occur, followed later by a use of x, we can frequently substitute y for x (provided neither has changed between the copy and the use).
  - This is another optimization that can be done locally with the sue of a DAG (or globally)
  - If x is a temporary, we have now created useless code. We discuss its removal in 6.1.4

- Reduction in Strength.
  - It is occasionally possible to replace an expensive operation with a cheaper one; this is known as *reduction in strength*.
    - An integer that is to be multiplied by 2 can be shifted 1 bit to the left.
    - a = x**y (in FORTRAN) would normally be compiled as:
      - T1 := ln(x)
      - T2 := y * T1
      - a := exp(T2)
  - We will return to reduction in strength when we discuss loop optimization in Sec. 6.1.3

- Substitution of In-Line Code.
  - A statement like:
    - i := abs(j)
  - is at least nominally a function call.
  - In most computers it is easier simply to test the sign of j and complement it if it is negative.
  - May computers have an instruction that will simply force the contents of a register to be positive
    - L 3,J
    - LPR 3,3 Force a positive value
    - ST 3,I

- Common Subexpressions.
  - We saw the use of DAGs to detect common subexpressions locally in Chapter 5.
  - We will pursue this further in the next section, Loop Optimization (6.1.3) and then again in Section 6.2

1.3 Loop Optimization

- Most programs spend the bulk of their execution time in loops, and especially in the innermost levels of nested loops.
- There is a well-known saying that 10 percent of the code accounts for 90 percent of the execution time; this 10 percent is usually found in loops.
- Hence loop optimization is of particular importance.

- Loop-Invariant Expressions
  - A relative of common subexpression is the repeated subexpression.
    - for k:= 1 to 1000 do
      - c[k] := 2*(p-q)*(n-k+1)/(sqr(n)+n);
  - Certain parts of this expression are not dependent upon the loop variables (k) and should be moved out of the loop.
    - 2*(p-q)
    - sqr(n)+n
  - This process is called *invariant code motion*.

- There are other operations that could be moved that are not reachable by the programmer.

```
for i:= 1 to 30 do
    for j:= 1 to 30 do
        for k:= 1 to 30 do
            x[i, j, k] := y[i, j, k] + z[i, j, k]
```

- To find the address of [i,j,k] we must compute:
  - addr(x[i, j, k]) = base(x) + offset(i, j, k)
  - offset(i, j, k) = $e(k-l_3)$ + offset(i, j)
  - offset(i, j) = $e(j-l_2)(u_3+1-l_3)$+ offset(i)
  - offset(i) = $e(i-l_1)(u_2+1-l_2)(u_3+1-l_3)$
  - where u_i and l_i are the upper and lower bounds of each dimension.
  - Note: offset(i) and offset(i, j) are invariants to the inner loops.

- T1 := (u3 + 1 - l3);
- T2 := u2 + 1 - l2);
- for i := 1 to 30 do
-     begin
-     T3 := e*(I-l1)*T1*T2 {T3 is offset(i)}
-     for(j:= 1 to 30 do
-         begin
-         T4 := T3 + e*(j-l2)*T1 {T4 is offset(i, j)}
-         for k:= 1 to 30 do
-             begin
-             T5 := T4 + e*(k-l3);
-             x[T5] := y[T5] + z[T5]
-             end;
-         end;
-     end;

- The restrictions on high-level languages generally forbid us writing thing like this,
- But the optimizer, having access to the 3AC instructions that accomplish these computations, can recognize the loop invariants and shift them to the preheaders of their respective loops, as this code suggests.

- Reduction in Strength.
  - A particularly important kind of strength reduction is associated with loops.
    - for i:=1 to 1000 do
    -     sum:= sum + a[i];
  - Suppose the elements of a are 4 byte reals
    - i := 1
    - T1 := i - 1
    - T2 := 4 * i
    - T3 := a[T2]
    - …
    - i := i+1
  - We could change the 4*i statement to a 2-bit left shift

- Loop Unrolling.
  - The management of a loop normally requires a certain amount of initial setup and a certain amount of checking every time the loop is repeated.

```
for i := 1 to 20 do
   begin
   for j := 1 to 2 do
      write (x[i, j]:9:3);
   writeln;
   end;
```


- For the outer loop the overhead is acceptable, for the inner, it is not. So you may just duplicate the code and substitute j with 1 or 2.
- If you wish to include this kind of optimization, you must decide how much space you are willing to lose in order to save execution time.
- The optimizer must then figure the space/time trade-off for unrolling any particular loop and unroll it only if the trade-off is acceptable.

1.4 Global Optimization

- There are other optimizations that span more than a single basic block but are not associated with loops.

- Dead Code Elimination.
  - Dead code is code that is never executed r that does nothing useful.
  - It does not appear regularly, but it may result from copy propagation, or from the programmer's debugging variables.

- In the first case
  - T1 := k
  - …
  - x := x + T1
  - y := x - T1
  - …
- Which could be changed to
  - x : = x + k
  - y : = x - k
- If T1 has no other use, it is now dead, so the tuple with the assignment to it can be removed.

- In debugging a program, we will write
  - if trace then
  - begin
  - writeln(…);
  - writeln(…);
  - end;
- with
  - const trace = true;
  - when we are debugging
- Then when we are done we change it to
  - trace = false;
- Now the if is dead code, and should be removed.

- Sometimes the optimizer can get carried away.
  - for I := 1 to 128 do
  - begin
  - port[x] := 1;
  - delay(1); {do nothing for 1 millisecond }
  - port[x] : = 0;
  - delay(1);
  - end;
- An overly ambitious optimizer could remove the whole for loop.
- Therefore you may want a compiler flag to turn off certain optimizations over a certain piece of code.

- Code Motion.
  - Another form of code motion is when we are optimizing the program for size rather than for speed is *code hoisting*.
    - case p of
    - 1: c : = a + b*d;
    - 2: m := b*d - r;
    - 3: f := a - b*d;
    - 4: c:= q / (b*d + r)
    - end;
  - Observe the b*d in every option of the case statement.

- Such an expression is said to be very busy.
- Clearly we should be able to precompute b*d
  - T := b*d
  - case p of
  - 1: c : = a + T;
  - 2: m := T - r;
  - 3: f := a - T;
  - 4: c:= q / (T + r)
  - end;
- It is because of this moving-up that this particular kind of optimization is called hoisting.
- This may not save time, but it saves space.

2. DAGs Again

- We saw how to generate DAGs under the control of the parser in Chapter 5.
- If we generate 3AC directly under the parse, then we may have to construct a DAG from scratch during the optimization phase in order to detect and remove common subexpressions.

2.1 Generating DAGs from 3AC

- We can assume that every 3AC instruction is in one of two forms:
  - a := b **op** c
  - a := **op** b

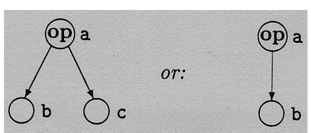


Figure 6.2

- Algorithm:
  - We assume there is an array curr that contains a pointer to the node that holds the variable's current value.
  - If curr[b] = nil,
    - create a leaf, label it b, make curr[b] point to this leaf.
    - If the instruction includes an operand c, do the same for it.
  - Look for an existing node labeled op that is already linked to the nodes pointed to by curr[b] and curr[c] (or curr[b] if that's the only operand).
    - If no such node exists, create one and link it.
    - In either case set curr[a] to this node.
    - If op is an assignment (a := b) we simply apply label a to the same node as b and make curr[a] point to it.
  - If another node in the DAG is labeled a, remove that label; in any case label the current node a.

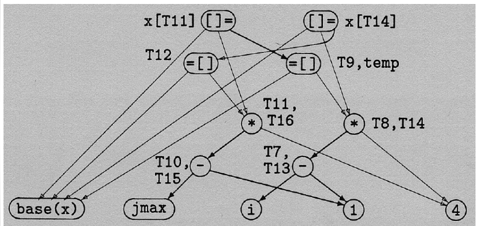


Figure 6.3(d)

- Constructing a DAG has to be done with some care, because we must be sure that two instances of an expression really refer to the same thing.
- We can cope with most problems by adding a flag to the record describing each node, indicating whether anything it refers to has been changed. This is sometimes known as a kill bit.
- If our optimizer includes data flow analysis between procedures, then we will know whether any variables are changed by the call and can avoid killing nodes unnecessarily.

2.2 Generating 3AC from DAGs

- The DAG, once we have it, tells us what the minimal code must be, and it also shows the dependencies among the various calculations.
- In generating 3AC from the DAG, we work from the leaves upward. Traversing the DAG in order of its dependencies is equivalent to topological sorting.

- Topological Sort Algorithm
  - Set n = 1
  - Repeat
    - Select any source, assign it the number n, and remove it and its incident edges from the graph
    - Set n = n+1
  - Until the graph is empty.
- You then go through this list in reverse order, starting with the last interior node in the list, and constructing the corresponding 3AC.

- In the example given in the text, which used the DAG shown a few slides ago, they cut the size of the block from thirteen 3AC statements to eight, and reduced the number of temporaries from ten to five.
- When there are no repeated subexpressions in a basic block, the DAG opens out into a tree.
- In this case you can traverse it in preorder and get similar results, but there is a better way.

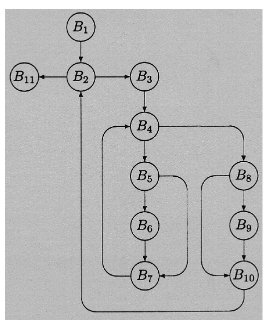
3. Data-Flow Analysis

- As soon as we expand the scope of optimization beyond the basic block, we must be able to trace how values make their way through a program.
- This process is called data-flow analysis.

3.1 Flow Graphs

- To show the transfer of control between basic blocks, we use a *flow graph*.
  - it is a directed graph in which each node is a basic block
  - the edges show the transfer of control between the blocks.

- When we isolated the basic blocks we used the branches in the program to determine their boundaries.
- In the flow graph, these branches are used to link the blocks



- The flow graph reveals the pattern of branching and looping within the procedure.
- Some Definitions
  - A block B_i is a *predecessor* of block B_j if there is an edge in the flow graph from B_i to B_j.
    - For example B_2 is a predecessor of B_3 (B_2 pred B_3)
  - We can also say that B_3 is a *successor* of B_2 (B_3 succ B_2)
    - Note that B_2 is not a predecessor of B_4
  - We say that B_i *dominates* a node B_j if we must pass through B_i when going from the start of the node to B_j
    - Node B_i is the *immediate dominator* of Node B_j if it's the last dominator we pass through before we reach Node B_j

◆ Dominance is a partial ordering, but, more importantly for our concerns, dominators can be displayed as a tree in which the root is their starting node and every interior node dominates all its descendants.

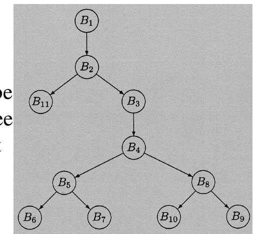


Figure 6.5

◆ The main usefulness of dominators is that they enable the compiler to recognize loops
  - There is a loop in the flow graph if there is an edge whose head dominates its tail.
    - Such an edge is called a *back edge*.
  - Recognizing loops is important, since loops tend to account for the bulk of the run time in any program and hence are prime candidates for optimization.
  - To recognize loops, we look for back edges
    - In our example flow graph we find two nested loops because there are two back edges
      - B_{10} to B_2
      - B_7 to B_4

3.2 Data-Flow Equations

◆ To optimize globally, we have to be able to track the movement of information between basic blocks.

◆ We must, in fact, consider all the blocks in the procedure because a computed value may be used in some remote block, not necessarily the successor of the block in which it was computed; thus we have to consider the procedure as a whole.

- We do this by setting up equations describing the fate of the computed values inside each block and describing how the interconnections of the blocks affect these values.

- These equations are the *data-flow equations*.
  - Data-flow analysis considers
    - what happens to variables or expression values within a block, that is, whether they are updated
    - how variables or values reach the beginning of a block
    - where they go from the end of a block

- Data Flow Analysis takes four different forms, depending on whether we work forward or backward through the flow graph and on whether we may consider any path to or from a block or must consider all paths.

- Each of these forms has its own set of data-flow equations.
  - We need an equation relating the conditions at a block boundary to those in adjacent blocks,
  - and we need an equation expressing how conditions change within the block.

- Fortunately, the equations for all four forms have the same general appearance, with minor variations, and the methods for solving them run along the same general lines.

- We also need some sort of initial conditions
  - for the beginning of the flow graph
    - (if our analysis is forward)
  - or for the end of the flow graph
    - (if our analysis is backward)

- We now look at the examples of the four types, develop equations, and then see how they are solved.

- Forward-Flow Analysis.
  - Consider the problem of eliminating common subexpressions.
    - We saw how to do this within a block; but that does not help us if we wish to remove redundant computations between blocks.
    - An expression is redundant if its value is already known; globally, it is redundant if its value will be already known **no matter how** we reach it.

- We will say that an expression is available at a point p if its value has been computed some place before p.
  - Therefore, an expression is available at the entry to a block if it is available at the outputs of all the predecessor blocks.
- Now for some definitions:
  - Let $in_a[B_i]$ be the set of all expressions available coming into any block B_i
  - Let $out_a[Bi]$ be the set of all those available at the output.
  - Then we may write:

$$in_a[B_i] = \bigcap_{B_j \text{ pred } B_i} out_a[B_j].$$

- Within a block, we say that an expression is *killed* if any of its operands are changed, since that means the expression's current value is no longer valid.
- And we say that an expression is *generated* if its value is computed in the block and not killed before the end of the block.
- If we let $gen[B_i]$ and $kill[B_i]$ be the sets of expressions generated and killed in B_i, then we can say:

$$out_a[B_i] = gen[B_i] \cup (in_a[B_i] - kill[B_i]).$$

- These equations are typical of a forward-flow, all-paths data-flow problem.
- If we take as our initial condition the fact that nothing is available at the start of the program, or
  - $in_a[B_1] = \{ \}$
- we have a set of equations which applied to the entire flow graph, will tell us precisely what expressions are available at any block in the graph.

- There are also example problems for forward-flow, any-path problems.
  - use-definition chaining (ud chains).
    - the ud chain for any use of a variable is a list of all the definitions in the program that reach the use.

- Backward-Flow Analysis.
  - As an example of backward-flow analysis, consider the problem of identifying very busy expressions.
    - An expression is very busy if, in all paths going forward from the expression, it is always used (that is, referred to) before it is killed by redefinition (either by the expression itself, or any of its operands)

- So, if we now let $in_b[B_i]$ be the set of very busy expressions at the start of any block B_i, and $out_b[B_i]$ those at the end, then:

$$out_b[B_j] = \bigcap_{B_i \text{ succ } B_j} in_b[B_i].$$

  - Note: we must take the intersection because if one node below us says that it is not busy, then we can't say it is busy.
  - This also means that we must work backwards from the bottom of the flow graph

- The set of very busy expressions at the start of a block can be written as:

$$in_b[B_i] = used[B_i] \cup (out_b[B_i] - kill[B_i]).$$

- From this we see that identifying very busy expressions is a backward-flow, all-paths problem.
  - Since we are working backward, our initial condition must be sought at the end of the program.
  - So, if a graph has n blocks, we may write
    - $out_b[B_n] = \{ \}$

- There are also Backward-Flow, any-path problems:
  - Consider the problem of identifying live variables
    - a variable is live if its current value is going to be used again
  - Another problem is the construction of definition-use chains (du chains)
    - This tells us for any definition of a variable, all the uses of that variable reached by that definition.

- These four families of equations
  - forward-flow
  - or backward-flow
  - and all-paths
  - or any-path
- are our mechanism for tracing the movement of information through the program.

3.3 Solving Data Flow Equations

- There is clearly a family resemblance among all four kinds of data-flow analysis.
- There is also a family resemblance among the methods for solving them as well.
- The most common general solution proceeds iteratively:
  - Make an initial guess, and use the equations to refine the guess.

- Refinement Algorithm:
  - Make initial guess
  - Repeat
    - Save old_guess
    - equations to obtain new_guess
  - Until old_guess == new_guess
- We must now consider what our initial guess will be.
  - One value is provided by the initial conditions.
  - In the case of any-path problem we assume that all sets are empty and the algorithm will fill them up.
  - In the case of the all-path problem we assume that all sets are full and the algorithm will empty them

- The solutions to Data-flow equations are said to be conservative, in that they make the optimizer overly careful.
- In order to guarantee that optimization does not change what the program computes, however, we want a conservative solution.
- At issue here is a Type I versus Type II error.
  - We would rather the optimizer miss a possible optimization rather than mess up the program by doing something wrong.

4. Optimization Techniques Revisited

- We will now return to the nonlocal optimizations introduced earlier and reconsider them in the light of the information we obtain from data-flow analysis.

4.1 Loop Optimizations

- We will start by considering how we can use data-flow information to identify loop invariants that can be moved and to find induction variables and simplify their computation.

- Loop Invariants
  - A 3AC statement is a loop invariant if its operands
    - Are constant, or
    - Are defined outside the loop, or
    - Are defined by some other invariant in the same loop.
  - To identify loop invariants, you need only go through the body of the loop and apply these tests to every statement.
    - You can find where the operands are defined by their du-chains.
    - It may be necessary to go through the loop repeatedly until no more invariant statements can be found.

- Once the invariants have been found, we can consider moving them.
  - Not every invariant can be moved.
    - For Example:
      - begin
      - …
      - a:=5;
      - …
      - a:=6;
      - …
      - end;
    - These are invariants by our example, but clearly they can't be moved; otherwise the value of a won't be changed at the proper point in the loop.
    - So, for an invariant a:= b+c to be movable, it must be the only definition of a in the loop.

  - Furthermore, we have to be certain that the definition of a is the only one reaching outside the loop.
    - a :=0;
    - for i:= 1 to lim do
    - begin
    - …
    - a := 6;
    - …
    - end;
    - yy:
  - The value of a at point yy depends on whether the loop was executed at all (lim was positive).
    - hence we cannot move a:= 6 outside the loop.
    - note that these conditions can be checked with the aid of up and du-chains.

- It is also important to move the statements in such a way that they appear in the same order in the preheader as they did in the body of the loop.
- If the loop contains
  - x:= 5;
  - …
  - y:- x+1;
- Then clearly the assignment to x must still precede the assignment to y after these statements have been moved to the preheader.

- Induction Variables.
  - An induction variable is any variable in a loop whose value changes by a constant step size every pass through the loop.
  - The problem is in detecting these
    - If the loop variable is i, then the computation of an induction variable j generally takes the form j:= ai+b
      - where a is the step size, and b is some constant. (these must be loop invariants)
  - These variables arise most commonly in computing the locations of subscripted variables.

- It takes a good deal of searching and checking to identity all induction variables, but the time is well spent.
  - The basic search is for 3AC instructions whose right-hand sides are of the form a*i, where a is a loop invariant and i is the loop counter (or some other induction variable).
  - The left hand sides of these instructions are now induction variables, which can produce more….

4.2 Other Optimizations

- Constant Folding.
  - To do constant folding or propagation globally, we must check the ud-chains of each use of the expression in which we suspect an operand is a constant to ensure that it has the same value no matter how it got there.

- Copy Propagation.
  - Redundant copy statements local to a single basic block will be removed by the DAG.
  - For copy propagation between the blocks, we need to know whether the copy is really redundant.
  - Suppose the copy is p=q
    - It will be redundant if:
      - No other definitions reach that use and
      - Neither p nor q is redefined on the way to that use.
    - These requirements can be checked by yet another set of forward-flow, all-paths data-flow equations.

5. More Optimization Issues

- In this section the text touches briefly on other kinds of optimization that are even beyond the scope of introductory books.
- These include:
  - the analysis of data flow between a subprogram and the program that calls them
  - problems associated with pointers.

- Interprocedural data-flow analysis.
  - It is desirable to be able to trace the movement of data between a subprogram and the program that calls it.
    - This is called interprocedural data-flow analysis.
  - If we can do this, then when we are constructing a DAG for a block that includes a function call, we will know which if any nodes in the DAG must be killed.
  - and when we are optimizing a loop, we will know whether it is safe to move a function call outside the loop.

- Problems with pointers.
  - Finding whether a simple variable is changed by a subprogram is reasonably straightforward, but in the case of arrays and data objects accessed by pointers, the problem is more difficult.
  - The picture is further complicated by the fact that
    - x[i] and x[j] may refer to the same thing (if i==j)
    - and *p may refer to the same thing as x[j]
  - A data object that is known by two different names is said to be aliased.
    - There are methods for dealing with aliasing, with arrays, and with pointers, but they are beyond the scope of an introductory text.
    - Aho, Sethi, and Ullman [1986] is, as usual, the most comprehensive reference.

6. Machine-Dependent Optimization

- Machine dependent optimization falls into two main classes:
  - optimizing register assignments subject to the constraints imposed by the hardware
  - the particular target machine may have in its set a handy instruction that provides a convenient shortcut.
- The architecture of the target machine bears on optimization and code generation.

6.1 Registers and Instructions

- We will look at some examples of register optimization here, although detailed techniques for selection registers will have to await the discussion of code generation in Chapter 7.
- Examples of instruction use are harder to find, since machine architectures vary so widely.

- Registers.
  - As an example of register use, it should be clear that the most frequently accessed variables should be held in registers instead of memory.
  - But there are never enough registers.
    - The first computers had one or perhaps two working registers (the second was for double length math operations)
    - The IBM System/360 was the first major architecture to provide multiple working registers.
      - They say that Iverson, the architect of the 360 family, when asked why he provided 16 general registers, replied, “Because I couldn’t manage 32.”

- The problem is further complicated by the fact that there are nearly always restrictions on the use of these registers.
  - Multiplication/Division with even/odd register pairs.
  - Stack Pointer,
  - Indirect Addressing restrictions,
  - …
- Restrictions like these handicap the programmer.
  - With many machines, it is not a question of optimization but of making the best of a bad deal.

- Instructions.
  - Examples of instruction use depend upon the target machine.
  - One common case, however, arises from the statement i = i+1, where i is an integer.
    - Unoptimized code will look like
      - L 3,I
      - A 3,=F’1’
      - ST 3,I
    - But almost any computer now has an instruction for incrementing a register.

- Many architectures provide instructions that support loops.
  - unfortunately some architectures limit the scope of the loop (their conditional jumps can only go so far)
- As an example of a more obscure instruction, consider the xchg instruction in the 80*86 set.
  - This exchanges the contents of two register.
  - When you program a sort, you use a temp, the optimizer could eliminate this, by swapping them in registers with this instruction.

- This brings two questions in its train.
  - First, is it worth the trouble to program an optimizer to keep an eye out for opportunities to use an instruction like xchg?
    - It is often easier to use a small repertoire of common instructions and ignore those that are more rarely useful.
    - This is part of the bases for RISC architectures.
  - Second, if temp has been optimized out of existence, then what will happen when the user runs the program through a debugger and wants to observe the value of temp?
    - In an integrated environment, you may want to omit these types of optimizations.
    - It maybe decided, as a matter of policy to avoid such optimizations in the first place on the grounds that, in a certain sense, they change the meaning of the program.

6.2 Peephole Optimization

- Machine-dependent optimization must be done after the final code-generation phase.
- Once the program has been translated into machine language, the analytical techniques, like data-flow analysis, are generally too difficult to be practical.

- Instead we look at code through a small moving window that spans just a few instructions at a time, and we look for small-scale redundancies visible through this window.
  - Because of this tiny window, the process is known as *peephole optimization*.
- We will now look at some of the redundancies found and removed by peephole optimization.

- Jumps over Jumps.
  - In conditional jumps,
    - if the destination is some point beyond where we are right now, the compiler won't know yet how far away that will be. The easiest way out is to compile a conditional jump as a jump over a jump
      - jnz x1
      - jmp xx
      - x1: …
    - But suppose that xx turns out be be only, say, 95 bytes away. Then this was unnecessary. In this case the peephole optimizer will substitute
      - jz xx
    - for the two jumps.

- Redundant Loads and Stores.
  - Naïve translation of 3AC is likely to result in sequences in which a variable is stored in memory and then immediately loaded back into the register from which it came.
  - It is easy for the peephole optimizer to catch and remove redundant loads, especially if they come right after the corresponding store.

- Other simplifications.
  - These include:
    - reduction in strength (if not done earlier)
      - multiplication by a power of 2
    - algebraic simplifications, and
      - adding 0 or multiplying by 1
    - making use of special instructions.

7. Summary: How Much Optimization?

- Optimization is a huge topic, and researchers have devised an enormous variety of techniques.
- We have looked at some of the most important ones, but we have only scratched the surface.

- If we know all these techniques, how many of them are we going to use?
  - Presumably the ultimate optimizing compiler is going to se every single one of them;
  - But in practice we may settle for much less, particularly because some techniques may be effective only rarely.
  - One obvious approach is to restrict our optimizations to those in which the ratio of payoff to cost is highest.

- Memory limitations may also influence our decision.
  - Global optimizations like interprocedural data-flow analysis may be difficult or impossible unless the entire source program is in memory.
  - If the amount of memory is limited, we may omit this optimization.
- Another possibility is to write two versions of the compiler and market the more elaborate one as an "optimizing compiler"
  - Even the less elaborate version will usually include some minimal amount of optimization, however.

- We could possibly take three things as the minimum
  - Local optimization by means of DAGs
    - DAGs are easy to construct, and it is not difficult to produce 3AC from a DAG.
  - Loop Optimization
    - Loop optimization is likely to require more work, but the payoff is likely to be high, since programs tend to spend so much of their time in loops.
  - Peephole Optimization
    - Peephole optimization is relatively cheap, particularly if the size of the peephole is small and if we restrict the transformations to simple ones.

- Many commercial compilers attempt nothing more than this minimum.