

UNIT V CODE OPTIMIZATION

INTRODUCTION

- The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.
- Optimizations are classified into two categories. They are
 - Machine independent optimizations:
 - Machine dependant optimizations:

Machine independent optimizations:

Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

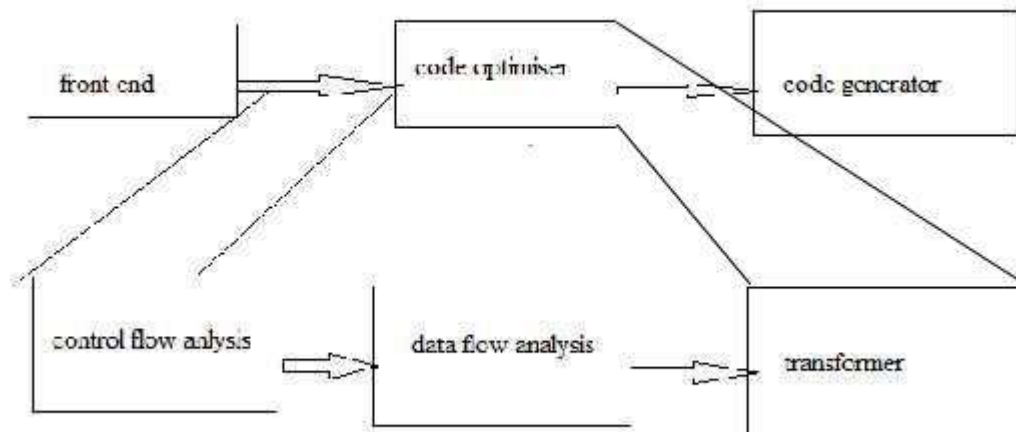
Machine dependant optimizations:

Machine dependant optimizations are based on register allocation and utilization of special machine-instruction sequences.

The criteria for code improvement transformations:

- ✓ Simply stated, the best program transformations are those that yield the most benefit for the least effort.
- ✓ The transformation must preserve the meaning of programs. That is, the optimization must not change the output produced by a program for a given input, or cause an error such as division by zero, that was not present in the original source program. At all times we take the “safe” approach of missing an opportunity to apply a transformation rather than risk changing what the program does.
- ✓ A transformation must, on the average, speed up programs by a measurable amount. We are also interested in reducing the size of the compiled code although the size of the code has less importance than it once had. Not every transformation succeeds in improving every program, occasionally an “optimization” may slow down a program slightly.
- ✓ The transformation must be worth the effort. It does not make sense for a compiler writer to expend the intellectual effort to implement a code improving transformation and to have the compiler expend the additional time compiling source programs if this effort is not repaid when the target programs are executed. “Peephole” transformations of this kind are simple enough and beneficial enough to be included in any compiler.

Organization for an Optimizing Compiler:



- Flow analysis is a fundamental prerequisite for many important types of code improvement.
Generally control flow analysis precedes data flow analysis.
Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs such as
 control flow
 graph Call graph
Data flow analysis (DFA) is the process of ascertaining and collecting information prior to program execution about the possible modification, preservation, and use of certain entities (such as values or attributes of variables) in a computer program.

PRINCIPAL SOURCES OF OPTIMISATION

A transformation of a program is called local if it can be performed by looking only at the statements in a basic block; otherwise, it is called global.

Many transformations can be performed at both the local and global levels. Local transformations are usually performed first.

Function-Preserving Transformations

There are a number of ways in which a compiler can improve a program without changing the function it computes.

- The transformations
 - ✓ Common sub expression elimination,
 - ✓ Copy propagation,
 - ✓ Dead-code elimination, and
 - ✓ Constant folding

are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations are performed.

- Frequently, a program will include several calculations of the same value, such as an offset in an array. Some of the duplicate calculations cannot be avoided by the programmer because they lie below the level of detail accessible within the source language.

➤ **Common Sub expressions elimination:**

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid recomputing the expression if we can use the previously computed value.

For example

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t4: = 4*i
t5: = n
t6: = b [t4] +t5
```

The above code can be optimized using the common sub-expression elimination as

```
t1: = 4*i
t2: = a [t1]
t3: = 4*j
t5: = n
t6: = b [t1] +t5
```

The common sub expression $t4: = 4*i$ is eliminated as its computation is already in $t1$.

And value of i is not been changed from definition to use.

➤ **Copy Propagation:**

Assignments of the form $f := g$ called copy statements, or copies for short. The idea behind the copy-propagation transformation is to use g for f , whenever possible after the copy statement $f := g$. Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x .

For example:

```
x=Pi;
.....
A=x*r*r;
```

The optimization using copy propagation can be done as follows:

```
A=Pi*r*r;
```

Here the variable x is eliminated

➤ **Dead-Code Eliminations:**

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point. A related idea is dead or useless code, statements that compute

values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations. An optimization can be done by eliminating dead code.

Example:

```
i=0;
if(i=1)
{
  a=b+5;
}
```

Here, „if“ statement is dead code because this condition will never get satisfied.

➤ **Constant folding:**

- We can eliminate both the test and printing from the object code. More generally, deducing at compile time that the value of an expression is a constant and using the constant instead is known as constant folding.
- One advantage of copy propagation is that it often turns the copy statement into dead code.
- ✓ For example,
 $a = 3.14157/2$ can be replaced by
 $a = 1.570$ there by eliminating a division operation.

➤ **Loop Optimizations:**

- We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- Three techniques are important for loop optimization:
 - ✓ code motion, which moves code outside a loop;
 - ✓ Induction-variable elimination, which we apply to replace variables from inner loop.
 - ✓ Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

➤ **Code Motion:**

- An important modification that decreases the amount of code in a loop is code motion. This transformation takes an expression that yields the same result independent of the number of times a loop is executed (a loop-invariant computation) and places the expression before the loop. Note that the notion “before the loop” assumes the existence of an entry for the loop. For example, evaluation of limit-2 is a loop-invariant computation in the following while-statement:

```
while (i <= limit-2)  /* statement does not change limit*/
```

Code motion will result in the equivalent of

```

t:= limit-2;
while (i<=t)  /* statement does not change limit or t */

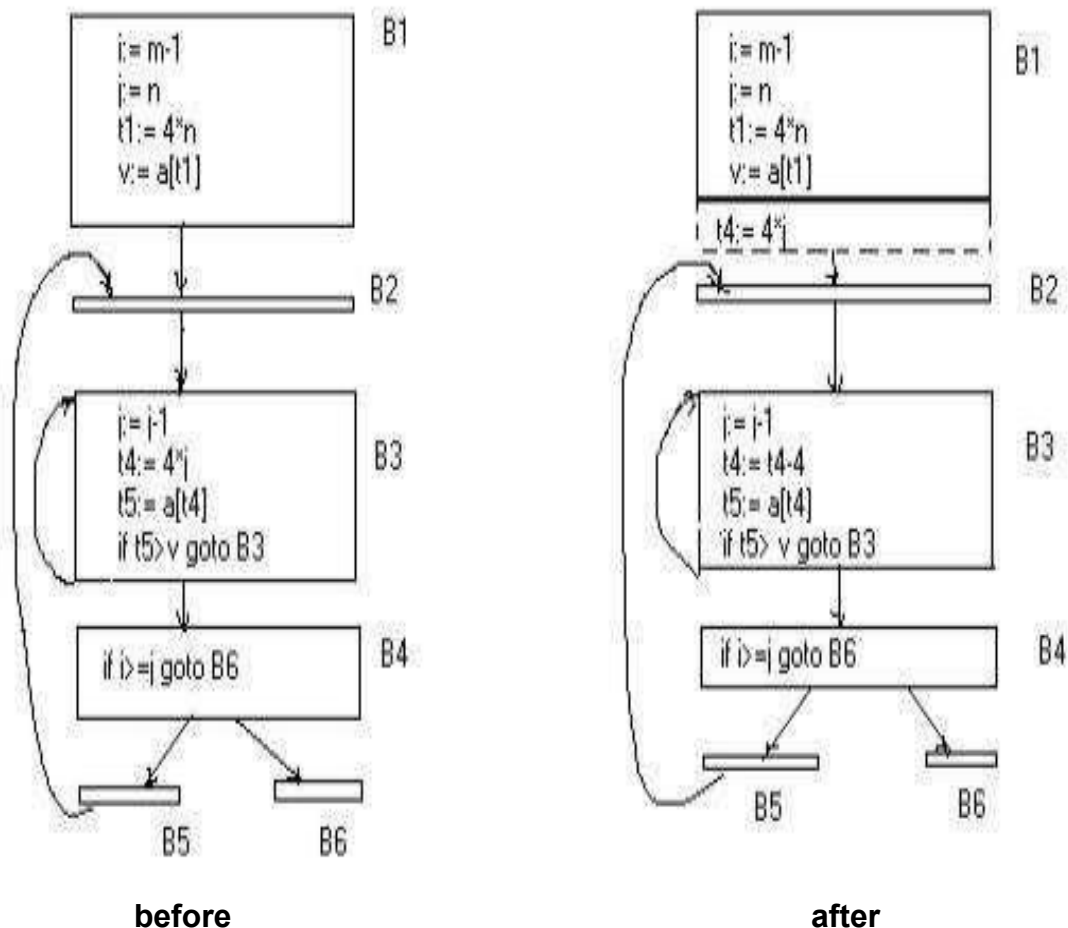
```

➤ Induction Variables :

- Loops are usually processed inside out. For example consider the loop around B3.
- Note that the values of j and t_4 remain in lock-step; every time the value of j decreases by 1, that of t_4 decreases by 4 because $4*j$ is assigned to t_4 . Such identifiers are called induction variables.
- When there are two or more induction variables in a loop, it may be possible to get rid of all but one, by the process of induction-variable elimination. For the inner loop around B3 in Fig. we cannot get rid of either j or t_4 completely; t_4 is used in B3 and j in B4. However, we can illustrate reduction in strength and illustrate a part of the process of induction-variable elimination. Eventually j will be eliminated when the outer loop of B2 - B5 is considered.

Example:

As the relationship $t_4 := 4*j$ surely holds after such an assignment to t_4 in Fig. and t_4 is not changed elsewhere in the inner loop around B3, it follows that just after the statement $j := j-1$ the relationship $t_4 := 4*j-4$ must hold. We may therefore replace the assignment $t_4 := 4*j$ by $t_4 := t_4-4$. The only problem is that t_4 does not have a value when we enter block B3 for the first time. Since we must maintain the relationship $t_4 := 4*j$ on entry to the block B3, we place an initialization of t_4 at the end of the block where j itself is



initialized, shown by the dashed addition to block B1 in second Fig.

- The replacement of a multiplication by a subtraction will speed up the object code if multiplication takes more time than addition or subtraction, as is the case on many machines.

➤ **Reduction In Strength:**

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators. For example, x is invariably cheaper to implement as $x*x$ than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

OPTIMIZATION OF BASIC BLOCKS

There are two types of basic block optimizations. They are :

- 9 Structure-Preserving Transformations
- 9 Algebraic Transformations

Structure-Preserving Transformations:

The primary Structure-Preserving Transformation on basic blocks are:

- ✓ Common sub-expression elimination
- ✓ Dead code elimination
- ✓ Renaming of temporary variables
- ✓ Interchange of two independent adjacent statements.

➤ **Common sub-expression elimination:**

Common sub expressions need not be computed over and over again. Instead they can be computed once and kept in store from where it's referenced when encountered again – of course providing the variable values in the expression still remain constant.

Example:

```
a: =b+c
b: =a-d
c: =b+c
d: =a-d
```

The 2nd and 4th statements compute the same expression: $b+c$ and $a-d$ Basic block can be transformed to

```
a: = b+c
b: = a-d
c: = a
d: = b
```

➤ **Dead code elimination:**

It's possible that a large amount of dead (useless) code may exist in the program. This might be especially caused when introducing variables and procedures as part of construction or error-correction of a program – once declared and defined, one forgets to remove them in case they serve no purpose. Eliminating these will definitely optimize the code.

➤ **Renaming of temporary variables:**

- A statement $t := b + c$ where t is a temporary name can be changed to $u := b + c$ where u is another temporary name, and change all uses of t to u .
- In this we can transform a basic block to its equivalent block called normal-form block.

➤ **Interchange of two independent adjacent statements:**

- Two statements

$t_1 := b + c$

$t_2 := x + y$

can be interchanged or reordered in its computation in the basic block when value of t_1 does not affect the value of t_2 .

Algebraic Transformations:

- Algebraic identities represent another important class of optimizations on basic blocks. This includes simplifying expressions or replacing expensive operation by cheaper ones i.e. reduction in strength.
- Another class of related optimizations is constant folding. Here we evaluate constant expressions at compile time and replace the constant expressions by their values. Thus the expression $2 * 3.14$ would be replaced by 6.28.
- The relational operators $<=$, $>=$, $<$, $>$, $+$ and $=$ sometimes generate unexpected common sub expressions.
- Associative laws may also be applied to expose common sub expressions. For example, if the source code has the assignments

$a := b + c$

$e := c + d + b$

the following intermediate code may be generated:

$a := b + c$

$t := c + d$

$e := t + b$

- Example:

$x := x + 0$ can be removed

$x := y ** 2$ can be replaced by a cheaper statement $x := y * y$

- The compiler writer should examine the language carefully to determine what rearrangements of computations are permitted, since computer arithmetic does not always obey the algebraic identities of mathematics. Thus, a compiler may evaluate $x*y-x*z$ as $x*(y-z)$ but it may not evaluate $a+(b-c)$ as $(a+b)-c$.

LOOPS IN FLOW GRAPH

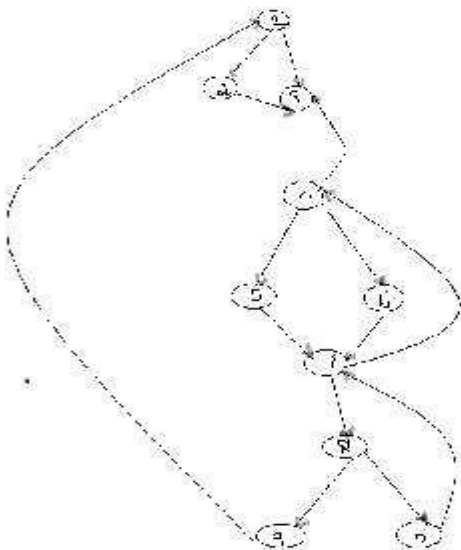
A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control.

Dominators:

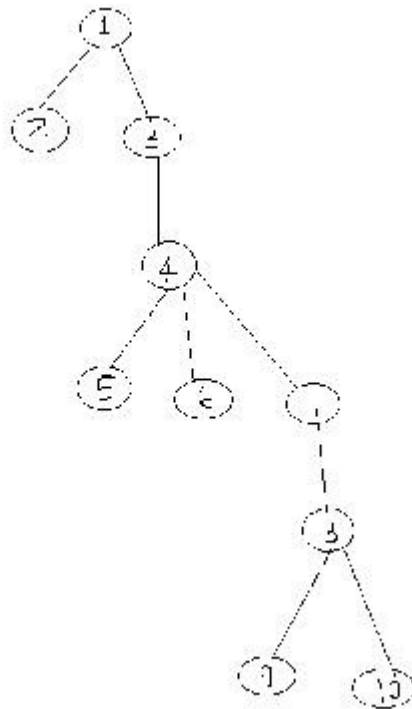
In a flow graph, a node d dominates node n , if every path from initial node of the flow graph to n goes through d . This will be denoted by $d \text{ dom } n$. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarly every node dominates itself.

Example:

- *In the flow graph below,
- *Initial node, node 1 dominates every node.
- *node 2 dominates itself
- *node 3 dominates all but 1 and 2.
- *node 4 dominates all but 1, 2 and 3.
- *node 5 and 6 dominates only themselves, since flow of control can skip around either by going through the other.
- *node 7 dominates 7, 8, 9 and 10.
- *node 8 dominates 8, 9 and 10.
- *node 9 and 10 dominates only themselves.



- The way of presenting dominator information is in a tree, called the dominator tree in which the initial node is the root.
- The parent of each other node is its immediate dominator.
- Each node d dominates only its descendants in the tree.
- The existence of dominator tree follows from a property of dominators; each node has a unique immediate dominator in that is the last dominator of n on any path from the initial node to n .
- In terms of the dom relation, the immediate dominator m has the property is $d \neq n$ and $d \text{ dom } n$, then $d \text{ dom } m$.



$$D(1) = \{1\} \quad D(2) = \{1, 2\}$$

$$D(3) = \{1, 3\}$$

$$D(4) = \{1, 3, 4\}$$

$$D(5) = \{1, 3, 4, 5\}$$

$$D(6) = \{1, 3, 4, 6\}$$

$$D(7) = \{1, 3, 4, 7\}$$

$$D(8) = \{1, 3, 4, 7, 8\}$$

$$D(9) = \{1, 3, 4, 7, 8, 9\}$$

$$D(10) = \{1, 3, 4, 7, 8, 10\}$$

Natural Loop:

- One application of dominator information is in determining the loops of a flow graph suitable for improvement.
- The properties of loops are
 - ☞ A loop must have a single entry point, called the header. This entry point dominates all nodes in the loop, or it would not be the sole entry to the loop.
 - ☞ There must be at least one way to iterate the loop (i.e.) at least one path back to the header.
- One way to find all the loops in a flow graph is to search for edges in the flow graph whose heads dominate their tails. If $a \rightarrow b$ is an edge, b is the head and a is the tail. These types of edges are called as back edges.

☞ Example:

In the above graph,

```
7 → 4    4 DOM 7
10 → 7    7 DOM 10
4 → 3
8 → 3
9 → 1
```

- The above edges will form loop in flow graph.
- Given a back edge $n \rightarrow d$, we define the natural loop of the edge to be d plus the set of nodes that can reach n without going through d . Node d is the header of the loop.

Algorithm: Constructing the natural loop of a back edge.

Input: A flow graph G and a back edge $n \rightarrow d$.

Output: The set loop consisting of all nodes in the natural loop $n \rightarrow d$.

Method: Beginning with node n , we consider each node $m \neq d$ that we know is in loop, to make sure that m 's predecessors are also placed in loop. Each node in loop, except for d , is placed once on stack, so its predecessors will be examined. Note that because d is put in the loop initially, we never examine its predecessors, and thus find only those nodes that reach n without going through d .

```
Procedure insert( $m$ );
if  $m$  is not in loop then begin
     $loop := loop \cup \{m\}$ ;
    push  $m$  onto stack
end;

 $stack := \text{empty};$ 
```

```

loop := {d};
insert(n);
while stack is not empty do begin
    pop m, the first element of stack, off stack;
    for each predecessor p of m do insert(p)
end

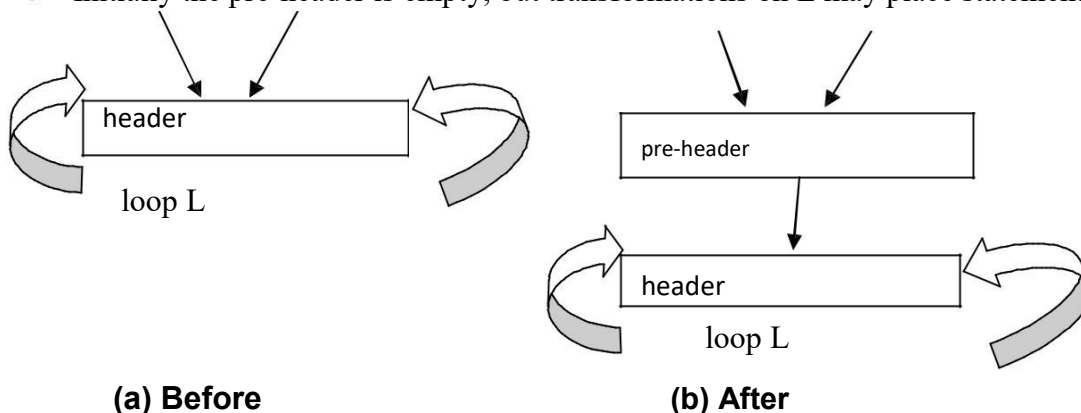
```

Inner loop:

- If we use the natural loops as “the loops”, then we have the useful property that unless two loops have the same header, they are either disjointed or one is entirely contained in the other. Thus, neglecting loops with the same header for the moment, we have a natural notion of inner loop: one that contains no other loop.
- When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.

Pre-Headers:

- Several transformations require us to move statements “before the header”. Therefore begin treatment of a loop L by creating a new block, called the preheader.
- The pre-header has only the header as successor, and all edges which formerly entered the header of L from outside L instead enter the pre-header.
- Edges from inside loop L to the header are not changed.
- Initially the pre-header is empty, but transformations on L may place statements in it.



Reducible flow graphs:

- Reducible flow graphs are special flow graphs, for which several code optimization transformations are especially easy to perform, loops are unambiguously defined, dominators can be easily calculated, data flow analysis problems can also be solved efficiently.
- Exclusive use of structured flow-of-control statements such as if-then-else, while-do, continue, and break statements produces programs whose flow graphs are always reducible.

- The most important properties of reducible flow graphs are that there are no jumps into the middle of loops from outside; the only entry to a loop is through its header.
- **Definition:**
A flow graph G is reducible if and only if we can partition the edges into two disjoint groups, *forward* edges and *back* edges, with the following properties.
 - ☞ The forward edges form an acyclic graph in which every node can be reached from initial node of G .
 - ☞ The back edges consist only of edges where heads dominate their tails.
 - ☞ Example: The above flow graph is reducible.
- If we know the relation DOM for a flow graph, we can find and remove all the back edges.
- The remaining edges are forward edges.
- If the forward edges form an acyclic graph, then we can say the flow graph reducible.
- In the above example remove the five back edges $4 \rightarrow 3$, $7 \rightarrow 4$, $8 \rightarrow 3$, $9 \rightarrow 1$ and $10 \rightarrow 7$ whose heads dominate their tails, the remaining graph is acyclic.
- The key property of reducible flow graphs for loop analysis is that in such flow graphs every set of nodes that we would informally regard as a loop must contain a back edge.

INTRODUCTION TO GLOBAL DATAFLOW ANALYSIS

- In order to do code optimization and a good job of code generation , compiler needs to collect information about the program as a whole and to distribute this information to each block in the flow graph.
- A compiler could take advantage of “reaching definitions” , such as knowing where a variable like *debug* was last defined before reaching a given block, in order to perform transformations are just a few examples of data-flow information that an optimizing compiler collects by a process known as data-flow analysis.
- Data-flow information can be collected by setting up and solving systems of equations of the form :

$$\text{out}[S] = \text{gen}[S] \cup (\text{in}[S] - \text{kill}[S])$$

This equation can be read as “ the information at the end of a statement is either generated within the statement , or enters at the beginning and is not killed as control flows through the statement.”

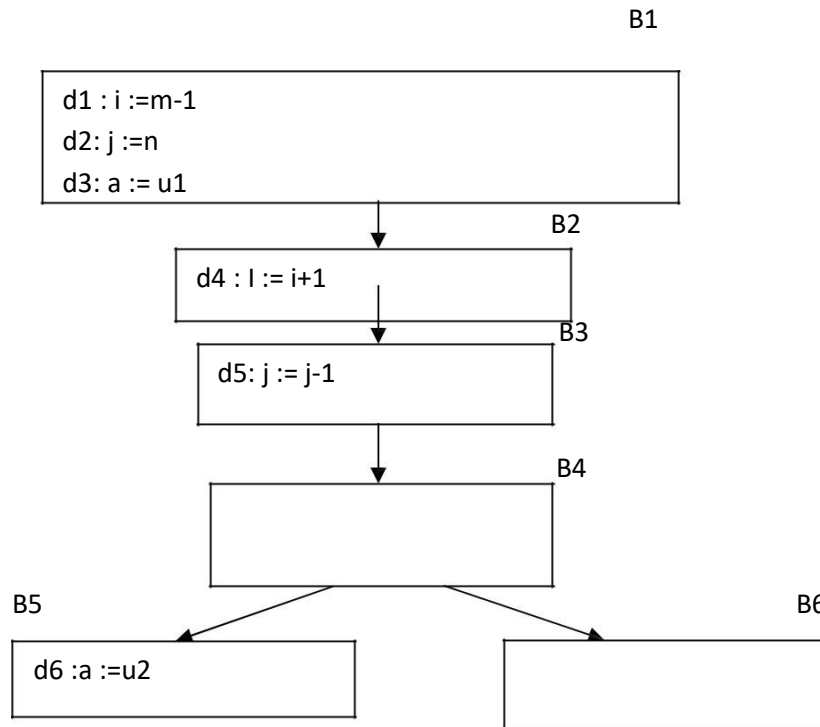
- The details of how data-flow equations are set and solved depend on three factors.
- ☞ The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining $\text{out}[s]$ in terms of $\text{in}[s]$, we need to proceed backwards and define $\text{in}[s]$ in terms of $\text{out}[s]$.

- ☞ Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.
- ☞ There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

Points and Paths:

- Within a basic block, we talk of the point between two adjacent statements, as well as the point before the first statement and after the last. Thus, block B1 has four points: one before any of the assignments and one after each of the three assignments.

www.studentsfocus.com



- Now let us take a global view and consider all the points in all the blocks. A path from p_1 to p_n is a sequence of points p_1, p_2, \dots, p_n such that for each i between 1 and $n-1$, either
 - \neg P_i is the point immediately preceding a statement and p_{i+1} is the point immediately following that statement in the same block, or
 - \neg P_i is the end of some block and p_{i+1} is the beginning of a successor block.

Reaching definitions:

- A definition of variable x is a statement that assigns, or may assign, a value to x . The most common forms of definition are assignments to x and statements that read a value from an i/o device and store it in x .
- These statements certainly define a value for x , and they are referred to as **unambiguous** definitions of x . There are certain kinds of statements that may define a value for x ; they are called **ambiguous** definitions. The most usual forms of **ambiguous** definitions of x are:
 - \neg A call of a procedure with x as a parameter or a procedure that can access x because x is in the scope of the procedure.
 - \neg An assignment through a pointer that could refer to x . For example, the assignment $*q = y$ is a definition of x if it is possible that q points to x . we must assume that an assignment through a pointer is a definition of every variable.
- We say a definition d reaches a point p if there is a path from the point immediately following d to p , such that d is not “killed” along that path. Thus a point can be reached

by an unambiguous definition and an ambiguous definition of the same variable appearing later along one path.

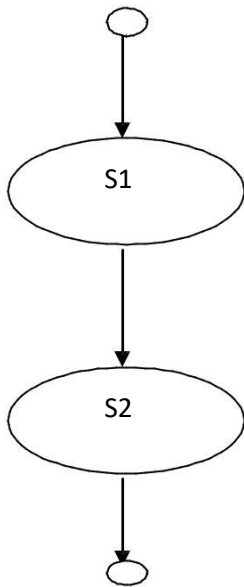
Data-flow analysis of structured programs:

- Flow graphs for control flow constructs such as do-while statements have a useful property: there is a single beginning point at which control enters and a single end point that control leaves from when execution of the statement is over. We exploit this property when we talk of the definitions reaching the beginning and the end of statements with the following syntax.

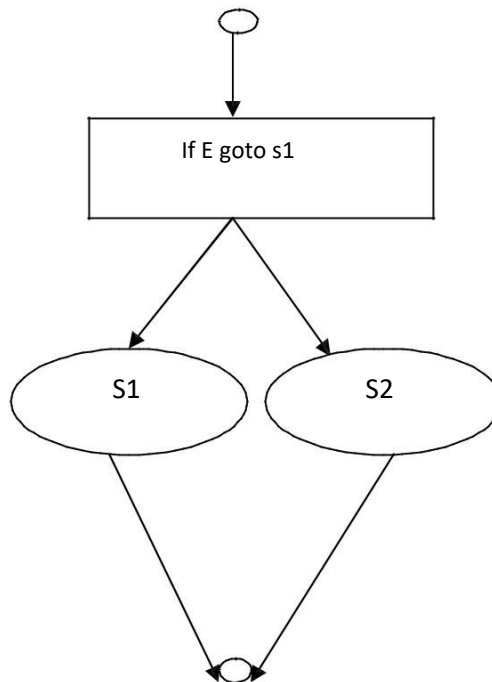
$S \longrightarrow \text{id} := E \mid S; S \mid \text{if } E \text{ then } S \text{ else } S \mid \text{do } S \text{ while } E$

$E \longrightarrow \text{id} + \text{id} \mid \text{id}$

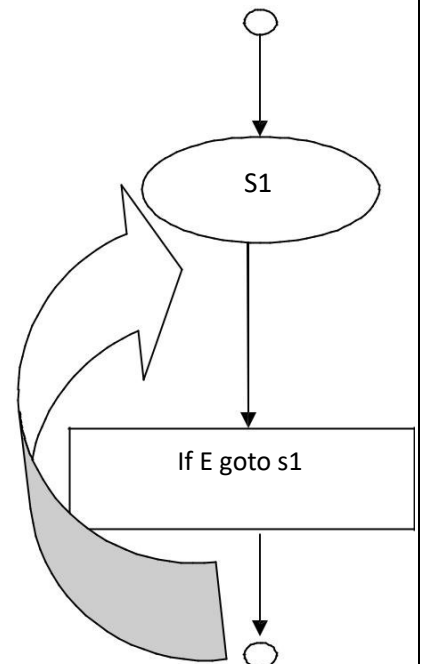
- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restricted forms.



S1 ; S2



IF E then S1 else S2

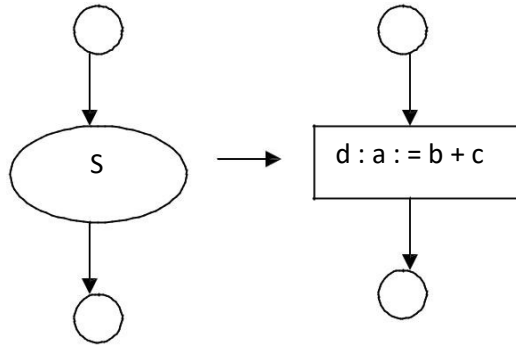


do S1 while E

- We define a portion of a flow graph called a *region* to be a set of nodes N that includes a header, which dominates all other nodes in the region. All edges between nodes in N are in the region, except for some that enter the header.
- The portion of flow graph corresponding to a statement S is a region that obeys the further restriction that control can flow to just one outside block when it leaves the region.

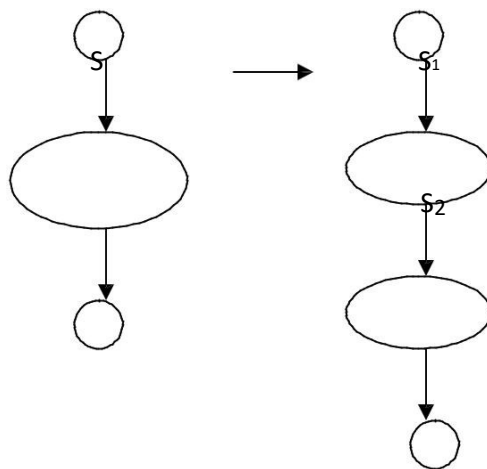
- We say that the beginning points of the dummy blocks at the entry and exit of a statement's region are the beginning and end points, respectively, of the statement. The equations are inductive, or syntax-directed, definition of the sets $in[S]$, $out[S]$, $gen[S]$, and $kill[S]$ for all statements S .
- **$gen[S]$ is the set of definitions "generated" by S while $kill[S]$ is the set of definitions that never reach the end of S .**
- Consider the following data-flow equations for reaching definitions :

i)



$$\begin{aligned} gen[S] &= \{ d \} \\ kill[S] &= D_a - \{ d \} \\ out[S] &= gen[S] \cup (in[S] - kill[S]) \end{aligned}$$

- Observe the rules for a single assignment of variable a . Surely that assignment is a definition of a , say d . Thus $Gen[S] = \{d\}$
- On the other hand, d "kills" all other definitions of a , so we write $Kill[S] = D_a - \{d\}$
Where, D_a is the set of all definitions in the program for variable a . ii)



$$\begin{aligned} gen[S] &= gen[S_2] \cup (gen[S_1] - kill[S_2]) \\ Kill[S] &= kill[S_2] \cup (kill[S_1] - gen[S_2]) \\ in[S_1] &= in[S] \\ in[S_2] &= out[S_1] \\ out[S] &= out[S_2] \end{aligned}$$

- Under what circumstances is definition d generated by $S=S_1; S_2$? First of all, if it is generated by S_2 , then it is surely generated by S . if d is generated by S_1 , it will reach the end of S provided it is not killed by S_2 . Thus, we write

$$\text{gen}[S] = \text{gen}[S_2] \cup (\text{gen}[S_1] - \text{kill}[S_2])$$
- Similar reasoning applies to the killing of a definition, so we have

$$\text{Kill}[S] = \text{kill}[S_2] \cup (\text{kill}[S_1] - \text{gen}[S_2])$$

Conservative estimation of data-flow information:

- There is a subtle miscalculation in the rules for gen and kill . We have made the assumption that the conditional expression E in the if and do statements are “uninterpreted”; that is, there exists inputs to the program that make their branches go either way.
- We assume that any graph-theoretic path in the flow graph is also an execution path, i.e., a path that is executed when the program is run with least one possible input.
- When we compare the computed gen with the “true” gen we discover that the true gen is always a subset of the computed gen . on the other hand, the true kill is always a superset of the computed kill .
- These containments hold even after we consider the other rules. It is natural to wonder whether these differences between the true and computed gen and kill sets present a serious obstacle to data-flow analysis. The answer lies in the use intended for these data.
- Overestimating the set of definitions reaching a point does not seem serious; it merely stops us from doing an optimization that we could legitimately do. On the other hand, underestimating the set of definitions is a fatal error; it could lead us into making a change in the program that changes what the program computes. For the case of reaching definitions, then, we call a set of definitions safe or conservative if the estimate is a superset of the true set of reaching definitions. We call the estimate unsafe, if it is not necessarily a superset of the truth.
- Returning now to the implications of safety on the estimation of gen and kill for reaching definitions, note that our discrepancies, supersets for gen and subsets for kill are both in the safe direction. Intuitively, increasing gen adds to the set of definitions that can reach a point, and cannot prevent a definition from reaching a place that it truly reached. Decreasing kill can only increase the set of definitions reaching any given point.

Computation of in and out:

- Many data-flow problems can be solved by synthesized translations similar to those used to compute gen and kill . It can be used, for example, to determine loop-invariant computations.
- However, there are other kinds of data-flow information, such as the reaching-definitions problem. It turns out that in is an inherited attribute, and out is a synthesized attribute depending on in . We intend that $\text{in}[S]$ be the set of definitions reaching the beginning of

S, taking into account the flow of control throughout the entire program, including statements outside of S or within which S is nested.

- The set $out[S]$ is defined similarly for the end of s. it is important to note the distinction between $out[S]$ and $gen[S]$. The latter is the set of definitions that reach the end of S without following paths outside S.

- Assuming we know $in[S]$ we compute out by equation, that is

$$Out[S] = gen[S] \cup (in[S] - kill[S])$$

- Considering cascade of two statements $S_1; S_2$, as in the second case. We start by observing $in[S_1]=in[S]$. Then, we recursively compute $out[S_1]$, which gives us $in[S_2]$, since a definition reaches the beginning of S_2 if and only if it reaches the end of S_1 . Now we can compute $out[S_2]$, and this set is equal to $out[S]$.

- Considering if-statement we have conservatively assumed that control can follow either branch, a definition reaches the beginning of S_1 or S_2 exactly when it reaches the beginning of S.

$$In[S_1] = in[S_2] = in[S]$$

- If a definition reaches the end of S if and only if it reaches the end of one or both sub statements; i.e,

$$Out[S]=out[S_1] \cup out[S_2]$$

Representation of sets:

- Sets of definitions, such as $gen[S]$ and $kill[S]$, can be represented compactly using bit vectors. We assign a number to each definition of interest in the flow graph. Then bit vector representing a set of definitions will have 1 in position I if and only if the definition numbered I is in the set.
- The number of definition statement can be taken as the index of statement in an array holding pointers to statements. However, not all definitions may be of interest during global data-flow analysis. Therefore the number of definitions of interest will typically be recorded in a separate table.
- A bit vector representation for sets also allows set operations to be implemented efficiently. The union and intersection of two sets can be implemented by logical or and logical and, respectively, basic operations in most systems-oriented programming languages. The difference $A-B$ of sets A and B can be implemented by taking the complement of B and then using logical and to compute A

Local reaching definitions:

- Space for data-flow information can be traded for time, by saving information only at certain points and, as needed, recomputing information at intervening points. Basic blocks are usually treated as a unit during global flow analysis, with attention restricted to only those points that are the beginnings of blocks.
- Since there are usually many more points than blocks, restricting our effort to blocks is a significant savings. When needed, the reaching definitions for all points in a block can be calculated from the reaching definitions for the beginning of a block.

Use-definition chains:

- It is often convenient to store the reaching definition information as “use-definition chains” or “ud-chains”, which are lists, for each use of a variable, of all the definitions that reaches that use. If a use of variable a in block B is preceded by no unambiguous definition of a , then ud-chain for that use of a is the set of definitions in $\text{in}[B]$ that are definitions of a . In addition, if there are ambiguous definitions of a , then all of these for which no unambiguous definition of a lies between it and the use of a are on the ud-chain for this use of a .

Evaluation order:

- The techniques for conserving space during attribute evaluation, also apply to the computation of data-flow information using specifications. Specifically, the only constraint on the evaluation order for the gen, kill, in and out sets for statements is that imposed by dependencies between these sets. Having chosen an evaluation order, we are free to release the space for a set after all uses of it have occurred.
- Earlier circular dependencies between attributes were not allowed, but we have seen that data-flow equations may have circular dependencies.

General control flow:

- Data-flow analysis must take all control paths into account. If the control paths are evident from the syntax, then data-flow equations can be set up and solved in a syntax-directed manner.
- When programs can contain goto statements or even the more disciplined break and continue statements, the approach we have taken must be modified to take the actual control paths into account.
- Several approaches may be taken. The iterative method works arbitrary flow graphs. Since the flow graphs obtained in the presence of break and continue statements are reducible, such constraints can be handled systematically using the interval-based methods

- However, the syntax-directed approach need not be abandoned when break and continue statements are allowed.