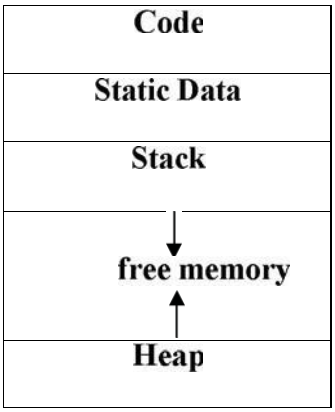
Unit-IV

STORAGE ORGANISATION

- The executing target program runs in its own logical address space in which each program value has a location.
- The management and organization of this logical address space is shared between the complier, operating system and target machine. The operating system maps the logical address into physical addresses, which are usually spread throughout memory.

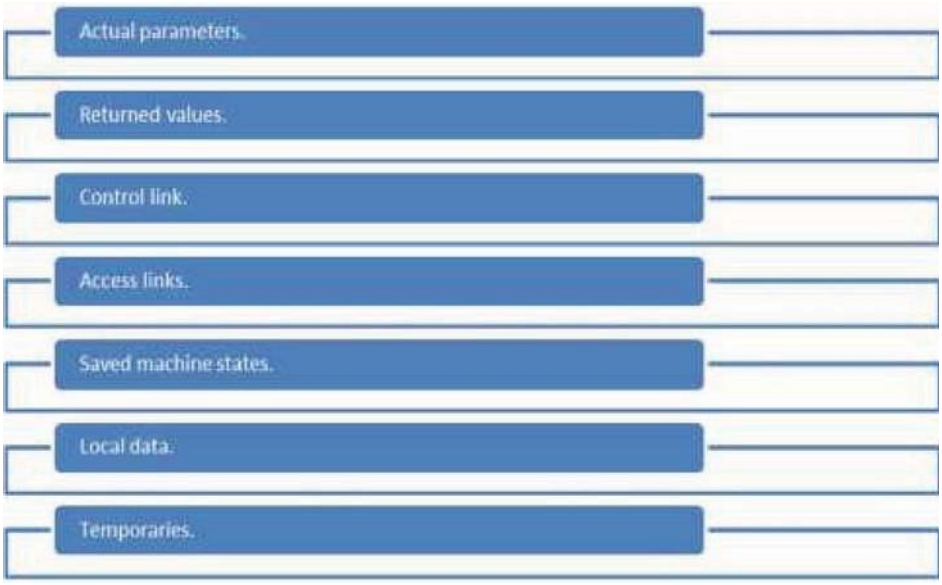


Run-time storage comes in blocks, where a byte is the smallest unit of addressable bytes and given the address of first byte.

- The storage layout for data objects is strongly influenced by the addressing constraints of the target machine.
- A character array of length 10 needs only enough bytes to hold 10 characters, a compiler may allocate 12 bytes to get alignment, leaving 2 bytes unused.
- This unused space due to alignment considerations is referred to as padding.
- The size of some program objects may be known at run time and may be placed in an area called static.
- The dynamic areas used to maximize the utilization of space at run time are stack and heap.

Activation records:

- Procedure calls and returns are usually managed by a run time stack called the *control stack.*
- Each live activation has an activation record on the control stack, with the root of the activation tree at the bottom, the latter activation has its record at the top of the stack.
- The contents of the activation record vary with the language being implemented. The diagram below shows the contents of activation record.



- Local data belonging to the procedure whose activation record this is.
- A saved machine status, with information about the state of the machine just before the call to procedures.
- An access link may be needed to locate data needed by the called procedure but found elsewhere.
- A control link pointing to the activation record of the caller.

Space for the return value of the called functions, if any. Again, not all called procedures efficiency.

- The actual parameters used by the calling procedure. These are not placed in activation record but rather in registers, when possible, for greater efficiency.

STORAGE ALLOCATION STRATEGIES

The different storage allocation strategies are :

1. **Static allocation –** lays out storage for all data objects at compile time
2. **Stack allocation –** manages the run-time storage as a stack.
3. **Heap allocation –** allocates and deallocates storage as needed at run time from a data area known as heap.

STATIC ALLOCATION

In static allocation, names are bound to storage as the program is compiled, so there is no need for a run-time support package.

- Since the bindings do not change at run-time, everytime a procedure is activated, its names are bound to the same storage locations.
- Therefore values of local names are *retained* across activations of a procedure. That is, when control returns to a procedure the values of the locals are the same as they were when control left the last time.
- From the type of a name, the compiler decides the amount of storage for the name and decides where the activation records go. At compile time, we can fill in the addresses at which the target code can find the data it operates on.

STACK ALLOCATION OF SPACE

- All compilers for languages that use procedures, functions or methods as units of user-defined actions manage at least part of their run-time memory as a stack.
- Each time a procedure is called , space for its local variables is pushed onto a stack, and when the procedure terminates, that space is popped off the stack.

Calling sequences:

- Procedures called are implemented in what is called as calling sequence, which consists procedure can continue its execution after the call.
- The code in calling sequence is often divided between the calling procedure (caller) and the procedure it calls (callee).
- When designing calling sequences and the layout of activation records, the following principles are helpful:

CODE OPTIMIZATION

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 optimizingcompilers.

Optimizations are classified into two categories. Theyare
Machine independentoptimizations:
Machine dependantoptimizations:

Machine independentoptimizations:

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

Machine dependantoptimizations:

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

The criteria for code improvementtransformations:

- Simply stated, the best program transformations are those that yield the most benefit for the leasteffort.
- 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 programdoes.
- 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 programslightly.
- The transformation must be worth the effort. It does not make sense for a compilerwriter

- 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 anycompiler.
- Flow analysis is a fundamental prerequisite for many important types of code improvement.
- Generally control flow analysis precedes data flowanalysis.
- Control flow analysis (CFA) represents flow of control usually in form of graphs, CFA constructs sucha
    - control flow graph
    - Callgraph
- Data flow analysis (DFA) is the process of ascerting 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 computerprogram

Function-Preserving Transformations

- There are a number of ways in which a compiler can improve a program without changing the function itcomputes.
- Thetransformations
    - Common sub expressionelimination,
    - Copypropagation,
    - Dead-code elimination,and
    - Constant folding, are common examples of such function-preserving transformations. The other transformations come up primarily when global optimizations areperformed.
- 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 expressionselimination:

- 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 usethe previously computedvalue.

Forexample
t1: =4*i
t2: =a [t1]
t3: =4*j
t4:=4*i
t5:=n
t 6: =b [t 4] +t 5

The above code can be optimized using the common sub-expression eliminationas t1:=4*i
t2:=a
[t1]t3:=4*j
t5:=n
t6: =b [t1] +t5
The common sub expression t 4: =4*i is eliminated as its computation is already in t1. And value of i is not been changed from definition to use.

CopyPropagation:

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 maynot appear to be an improvement, but as we shall see it gives us an opportunity to eliminatex.

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-CodeEliminations:

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 deadcode.

Example: i=0;
if(i=1)
{
a=b+5;
}
Here,,,if'statement is dead codebecausethis 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 constantfolding.

- One advantage of copy propagation is that it often turns the copy statement into deadcode.

Forexample,
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 thatloop.
- Three techniques are important for loopoptimization:
  codemotion, which moves code outside aloop;

Induction -variable elimination, which we apply to replace variables from innerloop.
Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by anaddition

CodeMotion:

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 followingwhile-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 aroundB3.
- Note that the values of j and t4 remain in lock-step; every time the value of j decreases by 1, that of t4 decreases by 4 because 4*j is assigned to t4. Such identifiers are called inductionvariables.

- 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 t4 completely; t4 is used in B3 and j inB4.
- 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.

LOOPS IN FLOWGRAPH

- 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:

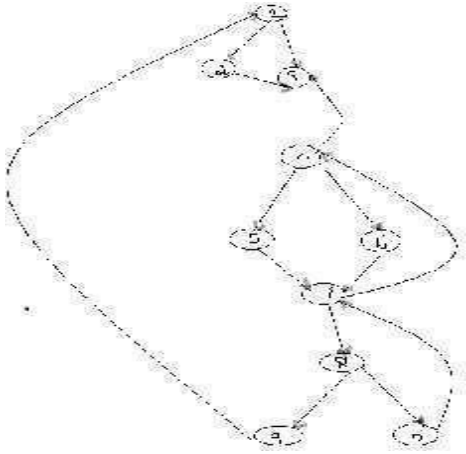
In a flow graph, a node d dominates node n, if every path from initial node of the flow graph to n goes through d. This will be denoted by *d dom n*. Every initial node dominates all the remaining nodes in the flow graph and the entry of a loop dominates all nodes in the loop. Similarlyeverynode dominates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

Example:

- In the flow graph below,
- Initial node,node1 dominates every node. *node 2 dominatesitself
- node 3 dominates all but 1 and 2. *node 4 dominates all but 1,2 and 3.

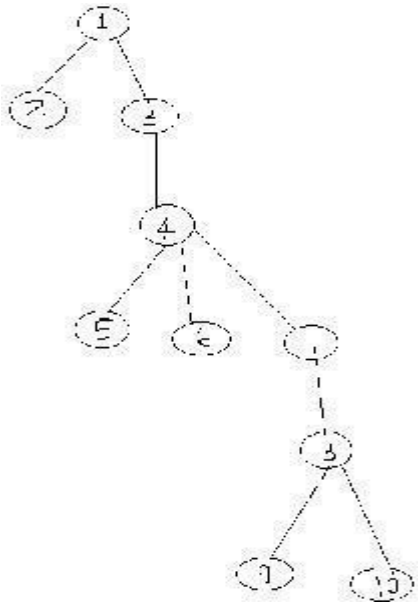
- node 5 and 6 dominates only themselves,since flow of control can skip around either by goin through theother.
- node 7 dominates 7,8 ,9 and 10. *node 8 dominates 8,9 and 10.

- 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 theroot.

- The parent of each other node is its immediatedominator.
- Each node d dominates only its descendents in thetree.
- 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 ton.
- In terms of the dom relation, the immediate dominator m has the property is d=!n and d dom n, then d domm.



D(1)={1}

D(2)={1,2}

D(3)={1,3}

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

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

(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}

NaturalLoop

- One application of dominator information is in determining the loops of a flow graph suitable forimprovement.
- The properties of loopsare
  - 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 theloop.
  - There must be at least one wayto 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→b is an edge, b is the head and a is the tail. These types of edges are called as backedg

Example:

In the above graph,

7 → 4 4 DOM7

0 →7 7 DOM10

4 → 3

8 → 3

9 →1

- The above edges will form loop in flowgraph.
- Given a back edge n → 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 theloop.

Algorithm: Constructing the natural loop of a back edge.
Input: A flow graph G and a back edge n→d

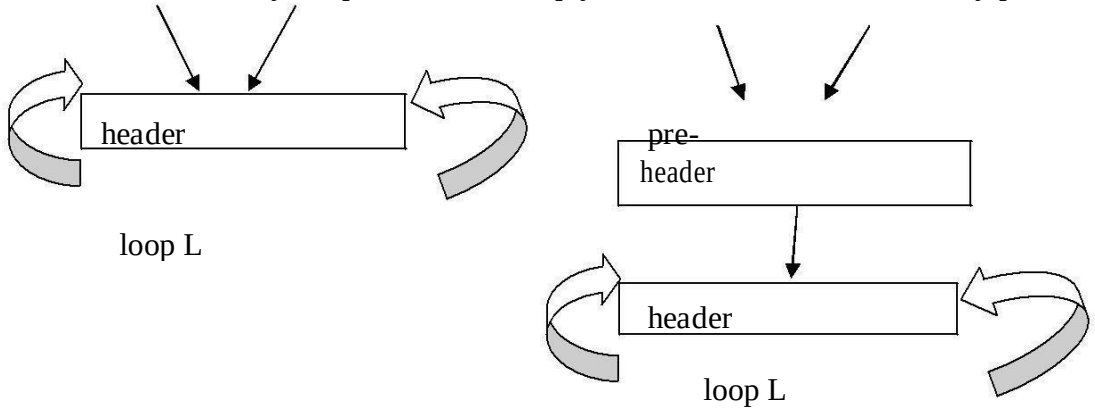
LOOP:

- 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 otherloop.
- When two natural loops have the same header, but neither is nested within the other, they are combined and treated as a singleloop.

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 thepreheater.

- The pre -header has only the header as successor, and all edges which formerly entered the header of Lfrom outside L instead enter thepre-header.
- Edges from inside loop L to the header are notchanged.
- Initially the pre-header is empty, but transformations on L may place statements init.



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 twodisjoint groups, *forward* edges and *back* edges, with the followingproperties.
- The forward edges from an acyclic graph in which every node can be reached from initial node ofG.
- The back edges consist only of edges where heads dominate theirstails.

Example: The above flow graph isreducible.

- If we know the relation DOM for a flow graph, we can find and remove all the back edges.
- The remaining edges are forwardedges.
- If the forward edges form an acyclic graph, then we can say the flow graphreducible.
- In the above example remove the five back edges 4→3, 7→4, 8→3, 9→1 and 10→7 whose heads dominate their tails, the remaining graph isacyclic.
- 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 backedge

PEEPHOLE OPTIMIZATION

- A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the targetprogram.
- A simple but effective technique for improving the target code is peephole optimization, a method for trying to improving the performance of the target program by examining a short sequence of target instructions (called the peephole) and replacing these instructions

by a shorter or faster sequence, wheneverpossible.

- The peephole is a small, moving window on the target program. The code in the peephole need not contiguous, although some implementations do require this.it is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.
    - We shall give the following examples of program transformations that are characteristic of peepholeoptimizations:

    - Redundant-instructionselimination
    - Flow-of-controloptimizations
    - Algebraicsimplifications
    - Use of machineidioms
    - UnreachableCode

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOVR0,a
(2) MOVa,R0

we can delete instructions (2) because whenever (2) is executed. (1) will ensure that the value of a is already in register R0.If (2) had a label we could not be sure that (1) was always executed immediately before (2) and so we could not remove (2).

INTRODUCTION TO GLOBAL DATAFLOWANALYSIS

- 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 flowgraph.
- 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-flowanalysis.
- Data- flow information can be collected by setting up and solving systems of equations of the form:

$$\underline{\text{out [S] = gen [S] U (in [S] – kill [S])}}$$

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 thestatement."

- The details of how data-flow equations are set and solved depend on threefactors.
- The notions of generating and killing depend on the desired information, i.e., on thedata flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms ofout[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 endpoints.
- There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to arrayvariables.

Data-flow analysis of structuredprograms:

- 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

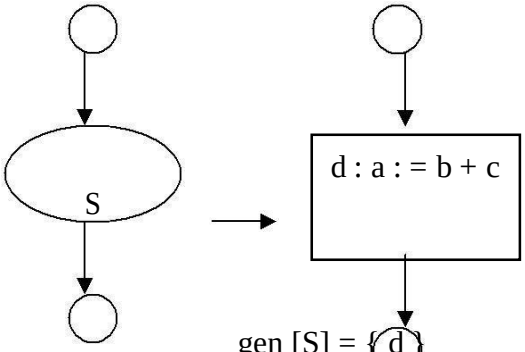
followingsyntax.

S id: = E| S; S | if E then S else S | do S
while E E id + id|id

- Expressions in this language are similar to those in the intermediate code, but the flow graphs for statements have restrictedforms.
- We define a portion of a flow graph called a *region* to be a set ofnodes 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 obeysthe 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 statementsS.
- gen[S] is the set of definitions "generated" by S while kill[S] is the set of definitions that never reach the end ofS.

Consider the following data-flow equations for reaching definitions:

i)

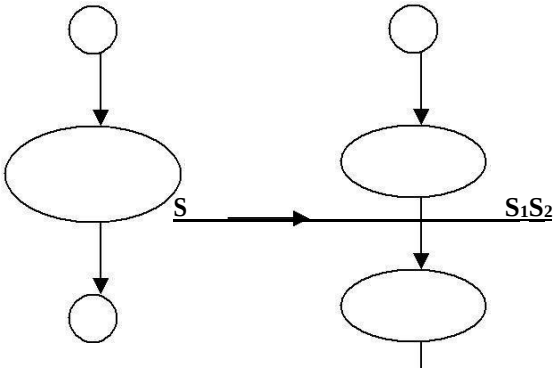


gen [S] = { d }
kill [S] = Da – { d }
out [S] = gen [S] U (in[S] – kill[S])

Observe the rules for a single assignment of variable a. Surely that assignment is a definition of a, say d. ThusGen[S]={d}
On the other hand, d "kills" all other definitions of a, so we write Kill[S]
= Da –{d}

Where, Da is the set of all definitions in the program for variable a.

ii



gen[S]=gen[S2] U(gen[S1]-kill[S2])
Kill[S] = kill[S2] U (kill[S1] – gen[S2])
in [S1] = in [S] in [S2] =
out [S1] out
[S] = out [S2]

- Under what circumstances is definition d generated by S=S1; S2? First of all, if it is generated by S2, then it is surely generated by S. if d is generated by S1, it will reach the end of S provided it is not killed by S2. Thus, wewrite

gen[S]=gen[S2] U (gen[S1]-kill[S2])
Similar reasoning applies to the killing of a definition, so we have Kill[S]
= kill[S2] U (kill[S1] –gen[S2])