# CD - UNIT - 5

## RUNTIME ENVIRONMENTS

## ① Stack allocation of space :

(i) **Activation Tree :-**

→ An activation tree is used to determine the way the control flows between procedures.

~~→ During flow of control~~

→ During Program execution, the control flow is sequential among procedures.

→ In a procedure, the execution begins at the starting point of its body.

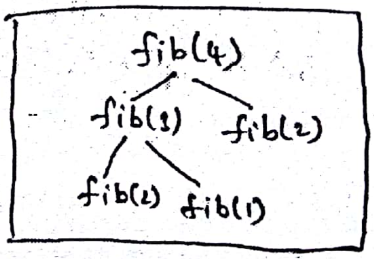
→ At the end of the procedure, control is returned to the calling instruction.

→ Each execution of the procedure is called an activation.

→ The activation tree depicts how control enters and leaves activations.

→ Activation tree for the Procedure Call fib(4) is shown in fig.

→ The activation tree is a graph of function calls.



## (ii) Activation Record:-

→ Activation Record (or) frame is a contiguous block of storage that contains all the information required for a single execution of a Procedure

→ An Activation Record consists of seven fields shown in fig.

Temporary Values

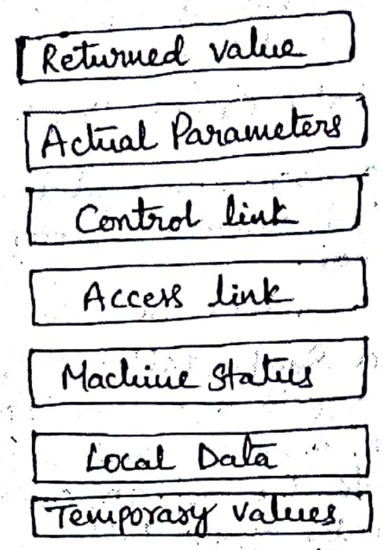
→ This field holds all the temporary values that are used in the evaluation of expressions.

Local Data

→ This field holds all the data that is local to a procedure.

Machine Status

→ This field saves the status of the machine when a call to Procedure occurs.



An Activation Record

Access link

→ This is an optional field.

→ This field holds a link to data (non-local) held in other activation records.

Control link

→ This is also optional field that contains a link to caller's activation record.

Actual Parameters

→ This field contains the actual parameters that are used by the calling procedure to pass them to the called Procedure.

Returned Value

→ This field contains a return value that is used by the called procedure to return to the calling procedure.

## 2 Access to Non Local Data on the Stack

**Q14. Explain how allocation strategies can access non local variable.**

**Answer :** Model Paper-II, Q6(a)

A nonlocal variable is a variable which is defined locally in a procedure.

**The Scope Rules of a Language**

The **lexical-scope** or **statical-scope rule** determines the declaration of a name based on the program text alone. Examples of languages that use this scope rule are Pascal, C and Ada.

The **dynamic-scope rule** determines the declaration of a name on sun time based on the current activation records. The languages Lisp, APL and Snobol use this scope rule.

**Lexical-scope in C**

'C' does not allow nested procedure declarations, therefore the lexical-scope rules for C are simple. In a C program the declaration of variables is followed by procedure definitions (procedures in C are called functions). A function is said to have a reference to a non local name *x* iff *x* is declared outside any function. The scope of *x* is within the functions that follow its declaration. There is a hole if *x* is redeclared with in function.

For example consider the following 'C' program.

```
int x;
f1( )
{
x = x – 10;
}
f2( )
{
   - - - -
}
f3( )
{
   x = x + 10;
}
main( )
{
   x = 10;
}
```

In this example '*x*' is declared outside all the functions, therefore its scope is in all functions.

The stack allocation strategy used for local names can also be used for lexically-scoped languages that donnot allow nested procedures. A name declared outside any procedure is a non local storage for all non local names memory is allocated statically at compile time. As we know the position of this storage, when a procedure references a non local name we simple use the static address of that name to access its value.

All other names are local to a procedure whose storage is within the current activation record which is at the top of the stack and accessible through the *top* pointer. This scheme does not work with nested procedures because an access to a non local may refer to data deep in the stack.

An advantage of static allocation for non locals is that, procedures can be passed as parameters and returned as results. In the absence of nested procedures a name which is non local to one procedure is non local to all procedures. Therefore, any procedures regardless how is it activated can use static address of that non local. When a procedure is returned as a result, the non locals of that procedure can be accessed similarly because they are bound to store allocated memory statically for them.

**Lexical-scope in Pascal**

Pascal allows the nested procedure declaration. That is, it is possible to define one procedure inside the other. In Pascal, the scope of a non local name *x* is in the scope of the most closely nested declaration of *x*.

For example, consider the following Pascal program for sorting an array.

```
Program sort(in, out);
    var x : array[0..10] of integer;
        t : integer;
    procedure insert;
        var i : integer;
        begin
            - - - - -
            x
            - - - -
            end {insert};
    Procedure swap(i, j : integer);
        begin
            t : = x[i];
        x[i] : = x[j];
        x[j] : = t
    end { swap };
```

Scanned by CamScanner

```
procedure quick sort(y, z : integer);
    var l, m : integer;
    function split(a, b : integer) : integer;
    var i. j : integer;
       begin
          -----
             x
          -----
             m
          -----
          swap(i, j);
        end { split};
    begin
        -----
    end{quicksort};
begin
    -----
end{sort}
```

In the above program sort consists of three procedures. They are, insert, swap and quick sort. The function split is nested within the procedure quick sort.

The function split references a non local *x*. According to most closely nested rule it is declared in the main program sort procedure. This rule also applies to procedure names, the procedure swap called by split is non local to split because it is declared in the main program sort.

To implement lexical scope we define nesting depth of a procedure. The nesting depth of the main program is, each time we go from an enclosing to an enclosed procedure the nest depth is incremented by 1. Therefore, the nesting depth for above program is given as,

sort : nesting depth 1

quick sort : nesting depth 2

split : nesting depth 3

We identify the nesting depth of a name from the nesting depth of the procedure in which it is declared. The nesting depth of names *i*, *x* and *m* defined is split in to 1, 2 and 3 respectively.

## 3 Heap Management :

**Q15. What are the two basic functions performed by memory manager? Also list the properties desired for a memory manager.**

**Answer :**

The two basic functions of memory manager include,

1. Memory allocation
2. Memory deallocation.

**1. Memory Allocation**

In this the memory manager maintains a contiguous chunk of free space of memory in heap storage. This memory is allocated to the program which has made a memory request for a variable or object. If the allocated size of variable is available in heap then allocation is done directly. Otherwise, the needed size is made available by increasing storage space. This can be done by getting sequential bytes of virtual memory from operating system. If the space is completely filled up, then this information (about no free space is available) is redirected to the application program by the memory manager.

**2. Memory Deallocation**

In this the memory manager returns the deallocated space back to the pool of free space. This free space get mixed up with the already available free space in heap. This deallocated memory can later be reused for other allocation requests.

**Properties of Memory Manager**

Properties of memory manager are as follows,

(i) **Maintaining Space Efficiency**

Space efficiency can be obtained by reducing the fragmentation of the program. To achieve this, the memory manager minimizes the amount of total heap space required by the program. Due to which a larger program can run on fixed virtual address space.

(ii) **Maintaining Program Efficiency**

Program efficiency can be achieved if the memory manager efficiently utilizes the available subsystem memory. This utilization helps to run the program at a faster rate.

(iii) **Minimizing Over Head**

The overhead can be minimized if allocation and deallocation operation are performed efficiently. As these are the frequent operations performed by memory manager.

CD 5-2

## CODE GENERATION

4 Issues in the Design of Code Generation

**Q16. Explain the issues in the design of code generator.** Nov.-12, Set-2, Q8(b)

**OR**

**Write in detail about the issues in the design of a code generator.** Nov.-12, Set-4, Q8(a)

**OR**

**What are the issues in code generation process? Explain in detail.**

**Answer :** (Model Paper-III, Q8(a) | Nov.-11, Set-2, Q8(a))

The following are the design issues of a code generator,

**1. Input**

The input to the code generator are the intermediate representation of the source code and the information in the symbol table. The symbol table gives the run-time addresses of names used in the intermediate representation.

The intermediate code generated by the code generator can be represented in several ways. These are linear representations such as postfix notation, graphical representation such as syntax trees and dags. Virtual machine representations such as stack machine code and three-address representations such as quadruples.

It is assumed that before the code generation phase begins, the source code is scanned, parsed and translated into intermediate representation by the front end. The values of names used in the intermediate representation are represented such that they can be directly manipulated by the target machine.

It is also assumed that the necessary type checking has been done and the input is free of errors. So, the code generation phase can proceed with these assumptions.

**2. Output**

The output of code generator is an object code. The object code can be produced in several forms. Various forms of object code are,

(a) Absolute machine code

(b) Relocatable machine code

(c) Assembly language code.

Each form of an object code has its own advantages and disadvantages.

**(a) Absolute Machine Code**

An absolute machine language program places the generated code in memory at some fixed location and executes the program immediately. For example, student-job compilers such as WATFIV, PASSGO and PL/C produce output as an absolute machine code. An advantage of absolute machine code is that the small programs can be compiled and executed quickly. But it can't call modules in other languages and compile subprograms separately. This form of object code is very fast compared to other forms.

**(b) Relocatable Machine Code**

Producing an output as a relocatable-machine language program allows compitation of subprograms separately. It also allows calling already compiled programs from an object module in other languages. After compiling a set of object modules, it requires a linking loader to link them together and load for execution. At little expense of linking and loading, which makes it slower we achieve great flexibility in computing subprograms separately.

**(c) Assembly Language Code**

An assembly language program makes the code generation process easier.

Assembly language code is mainly used in machines having less memory where complier requires several passes to execute them. This form of object code is slowest when compared to others.

**3. Memory Management**

The code generator along with front end performs mapping between the names (stored in the source program) and the addresses of data objects (stored in run-time memory). The entries in the symbol table names are created when their declarations are encountered while examining a procedure. The amount of storage required for a name depends on its data type. The relative address for the name can be determined from the symbol table information. When a name appears in a three-address statement, it refers to the symbol table entry for that name. The labels in three-address statements must be converted to addresses of instructions while generating the machine code.

4. **Selection of Machine Instructions**

It is difficult to select an instruction due to the nature of instruction set of the target machine. There are two important factors, the first is the uniformity and another is completeness of the instruction set. A target machine must support each data type in a uniform manner otherwise special handling is required to handle an exception to the general rule.

Other factors that make it difficult to select an instruction are instruction speeds and machine idioms. If the efficiency of the target program is not considered, then instruction selection is straight forward. For example, we create a code selection for each type of three-address statement that can be used to generate the target code for that construct. For example, if the form of three-address statement is as $a : b + c$ then, every such statement would be translated into the following sequence.

```
MOV b, R1
ADD c, R1
MOV R1, a
```

But, generating code as a statement-by-statement produces poor code.

The speed and size determines the quality of the generated code. A target machine with rich set of instructions can perform an operation in several ways. It is possible that an implementation generates the correct target code but it is inefficient. For example, three-address statement $x := x + 1$ (incrementing variable x by 1) can be more efficiently implemented as a single instruction INC (if the target machine has it in its instruction set) than implementing as the following sequence.

```
MOV x, R0
ADD #1, R0
MOV R0, x
```

The instruction speed also matters to produce good code sequences. But, it is often difficult to obtain accurate timing information. It is also difficult to determine which code sequence is best for the given three-address construct because this decision is based on the context where that context appears.

5. **Allocating Registers**

Machine instructions are small and faster in comparison to the memory instruction. This is because the machine instructions uses register while the memory instruction uses operands for its execution. Therefore, to generate a good code, the register must be efficiently utilized. To efficiently utilize registers, the following points must be considered.

❖ During register allocation, the variables that currently present in register should be selected.

❖ During register assignment, the specific register that will hold a variable should be selected.

However, such assignment is a tedious task assignment of registers to variables even in the case of single-register values. Further certain register-usage conventions are imposed by the hardware and/or the operating system of the target machine.

Another difficulty in register allocation is register-pairs (an even and next odd register) requirement of certain machines for some operands and results. For example, the IBM system/370 machine requires that the operations integer multiplication and integer division must involve register pairs.

6. **Evaluation Order**

The dramatic affect of evaluating the order of machine instruction is on the efficiency of the target code. Some evaluation order uses fewer registers than others to temporarily hold the intermediate results. Thus the choice of selecting best order of evaluation is difficult. The problem can be solved if target code for three-address statements is generated in the order in which they appear.

7. **Code Generation**

The most important issue of code generator is to produce a precise code. This issue is significant because a code generator might face a number of special cases. The main design goal is to design a code generator which is easy to implement, test and maintain.

## 5 The Target Language

**Q17. Explain in detail about a simple target machine model.**

**Answer :** Model Paper-IV, Q6(b)

**Simple Target Machine Model**

A target machine model containing $n$ general purpose registers i.e., $R_0, R_1, R_2, \ldots\ldots, R_{n-1}$ can model a three address machine using the instructions like,

1. Load
2. Store
3. Computation
4. Unconditional jumps
5. Conditional jumps.

1. **Load**

This operation, loads the value of one location to other location. This operation is of form LD location1, location2.

**Example**

(i) **LD destination, address**

The above instruction loads the value in address location to destination location.

(ii) **LD $r_1$, x**

The above instruction loads the value present in location $x$ to register $r_1$.

(iii) LD $r_1$, $r_2$

This type of instruction is called as register-to-register copy instruction. The above instruction loads the content of register $r_2$ to register $r_1$.

2. **Store**

This operation stores the content of one register to some desired location (say $x$). The store operation is of form,

ST location, register

**Example**

ST a, $r_2$

The above instruction stores the content of register $r_2$ into location $a$.

3. **Computation**

This operation performs various operations like addition, subtraction, multiplication and division on values present on source1 and source2 and later store the result on some specified location. The code operation is of form,

op destination, source1, source2

Here, 'op' specify operation.

**Example**

ADD $r_1$, $r_2$, $r_3$

The above instruction add the value present on $r_2$ and $r_3$ and later stores the result on $r_1$.

4. **Unconditional Jumps**

This instruction is of type,

BR L

Where, BR stands for branch and L stands for label.

The above instruction allow control to branch to machine instruction along with label.

5. **Conditional Jumps**

This instruction is of form,

B cond r, L

Here,

'cond' represents common test performed on values of register $r$.

$r$ stands for register and

$L$ stands for label.

## 6 . Addresses in the Target Code :

**Q18. Describe the code generation for simple procedure calls and returns using static allocation.**

**Answer :**

The process of code generation can be illustrated by the address statements like call, return, halt and action using the static allocation. The code generator determines the size and layout of activation records by using the name related information recorded in symbol table. In static allocation, the implementation of procedure calls having no arguments i.e., call callee can be performed using two target machine instructions as shown below,

ST callee.StaticArea #here + 20

BR callee.CodeArea

The *ST* instruction is responsible for storing the return address at the starting point of the activation record for the caller while the *BR* instruction is responsible for transfering control to the first entry in the target cock of the callee.

The attributes **callee.StaticArea** and **callee.CodeArea** are constants refering to the address of memory location holding the return address (i.e., the first location of the activation record) and the address of first entry of the called procedure in the code area respectively.

In the ST instruction, the operand #here + 20 represents the address of the instruction that appears immediately after BR instruction, which is nothing but the return address. The value, #here represents the address of current instruction and the symbol #represents immediate constant. Also, the sequence two instructions contains 3 constants. Thus, the entire sequence requires (2 + 3 = 5) – 5 words or 20 bytes. In general, caller do not exist for the first procedure. So in this case, HALT is the last instruction with a return control to the operating system. But, in case of procedure (other than the first one), its code termination occurs with a return control to the caller procedure. The implementation of return callee statement can be performed using BR instruction as follows,

BR *callee.staticArea.

This instruction is responsible for transfering control to the first location in the activation record of called procedure. Consider the following three address code

```
action 1          //code for main
call q
action 2
halt
action 3               //code for q
return.
```

The target code for the above three address code is given below,

```
1000 : ACTION1                    //code formain
1020 : ST 3064, # 1040 ....       //stored.return address 1040 in location 3064
1032 : BR 2000....                //call q
1040 : ACTION 2
1060 : HALT...                    //return to the operating system
2000 : ACTION 3...                //code for q
2020 : BR *3064
...
...
3000 :                            //The activation record for main is at location 3000 - 3063.
3004 :
3064 :                            //The activation record for q is at location 3064-4051.
3068 :
```

In the above target code, assume that procedures **'main'** and **'q'** starts at locations **1000** and **2000 respectively. Also,** assume that each instruction requires 20 bytes. Finally, assume that activation record for procedure **"main" is statically allocated** at location **3000** and for **'q'** procedure at location 3064.

The instruction that begins at location 1000 is responsible for implementing the following statements of main procedure.

Action 1;

Call q;

Action 2;

Halt;

At location **1020**, the **ST** instruction is responsible for storing the return address **1040** in the first location of activation record of called procedure **'q'**. Similarly, location **1032**, the **BR** instruction is responsible for transferring control to the target code of called procedure **'q'**. Once the execution of **ACTION 3** at location **2000** is completed, the **BR** instruction at location **2020** is executed. Now, ***3064** contains **1040** from the above call sequence at location **1020**. Once the code for procedure **'q'** ends, control returns to the called procedure i.e., at location **1040**. Now, the main procedure restarts its execution.

## 7 Storage allocation Strategies :

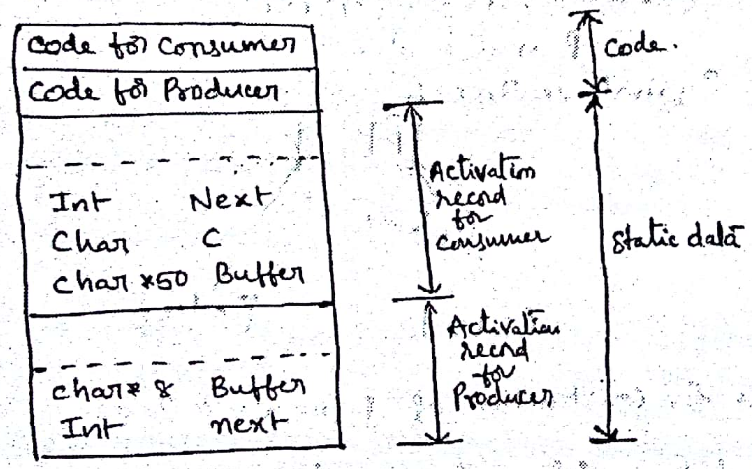
### (I) Static Allocation Strategy :-

→ Static Allocation strategy allocates the memory for all data objects at compile time.
→ The program variables are bound to the storage once it is allocated to them.
→ As the binding of variables for storage is done at compile time which do not change at runtime.
→ Therefore a runtime support package is not required.
→ This strategy implements static binding.
→ In static binding, each time a procedure is called the variables are bound to the same storage locations.

Ex:-

- FORTRAN allocates the memory for program variables of all subprograms irrespective of whether a subprogram is active (or) not.

- Static storage for Local Identifiers of Producer/consumer program is represented as



### (II) Stack Allocation Strategy :-

→ In this strategy, storage is allocated as a stack when a procedure is called for execution.
→ An activation record containing the information required for executing a procedure is pushed on the top of the stack.
→ Each time an activation record is pushed, the local variables of a procedure are bound to a new storage.
→ The values of local variables are lost when an activation record is popped out of the stack.
→ This happens when a procedure is terminated.
→ The memory space freed by popping an activation record can then be used to push another activation record.
→ A register stores value of the top of the stack.

8 **Basic Blocks:**

- A Basic Block is a collection of three-address statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.
- Every statement in a basic block is a three-address statement.

*Example:*

a=(b*c)/d+k

The three-address statements are,

```
t1:=b*c
t2:=t1/d
a:=t2+k
```

**Algorithm for Partitioning into Basic Blocks:**

- This algorithm is used to partition a sequence of three address statements into basic blocks.

**Input:** A sequence of three-address statements.

**Output:** A list of basic blocks with each three-address statement in exactly one block.

**Method:**

- We first determine the set of leaders, i.e, the first statements of basic blocks.
- The rules used are,
  i)The first statement is a leader.
  ii)Any statement that is the target of a conditional or unconditional goto is a leader.
  iii)Any statement that immediately follows a goto or conditional goto statement is a leader.
- A Basic Block is drawn for each leader followed by the set of statements.
- No Basic Block can have more than one leader.

9 **FLOW GRAPH:**

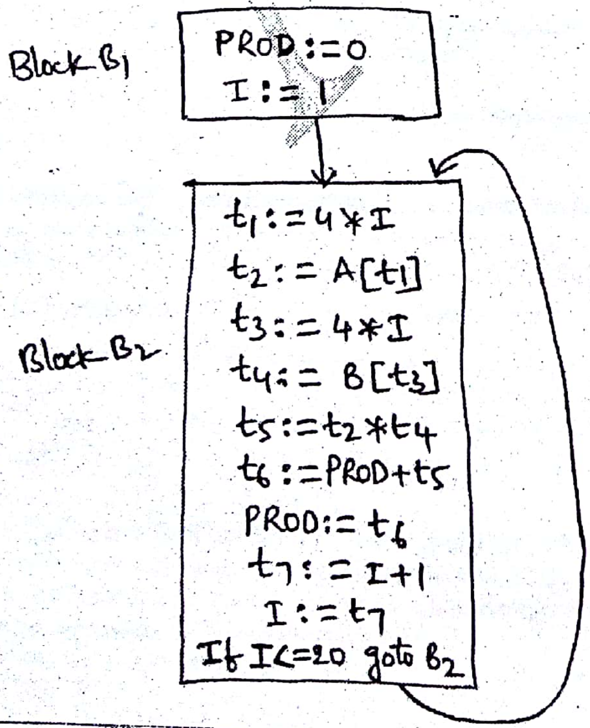
- A Flow Graph is a graphical representation of three-address statements.
- It is used to show the flow of control information to the set of basic blocks of a program.
- It consists of nodes and edges.
- A node of a flow graph is a basic block that performs some computations.
- An initial node consists of a block whose leader is the first statement.
- There is a directed edge from one node to another representing the flow of control between blocks.
- There is an edge from block B1 to block B2 if
  -In the execution sequence B2 immediately follow B1.
  -The last statement of B1 contains a conditional or unconditional jump to the leader(first statement) of B2.
- We say that predecessor of block B2 is block B1 and a successor of block B1 is block B2.

*Example:*

Consider the following program that finds the dot product of two vectors x and y of length 10.

```
begin
        PROD:=0;
        I:=1;
        do
            begin
                PROD:=PROD+A[I]*B[I];
                I:=I+1;
            end
        while I<=20
end
```

- The flow graph for the given program is constructed by first converting the program into three-address statements.
- The sequence of three-address statements are
  1. PROD:=0
  2. I:=1
  3. t1:=4*I
  4. t2:=A[t1]
  5. t3:=4*I
  6. t4:=B[t3]
  7. t5:=t2*t4
  8. t6:=PROD+t5
  9. PROD:=t6
  10. t7:=I+1
  11. I:=t7
  12. if I<:=20 goto 3
- By applying the algorithm for partitioning into basic blocks, the code contains statement 1 is a leader(rule 1) and statement 3 is a leader(rule 2).So the code contains two blocks.Then the flow graph for the code is



FLOW GRAPH

2.1. Loops in a FLOWGRAPH:

- Loop is a collection of nodes in a flow graph.
- Loops in a flow graph are of 4 types
  i)Dominators ii)Natural Loops iii)Inner Loops iv)Pre-headers

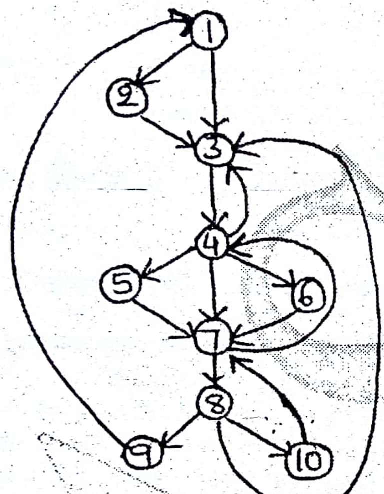
i) Dominators:

- A node 'D' is said to be dominator node, if it dominates another node 'N'.
- The general format to represent the dominators is

  D dom N

- D dom N means, in the flow graph every path from the initial node to N goes through the node D.
- Entry of a loop dominates all the nodes in the loop.
- Every node dominates itself.

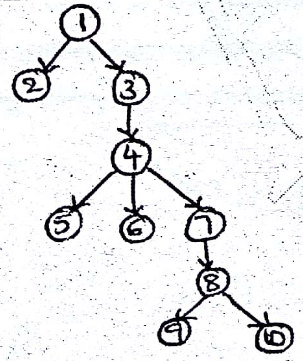
Flowgraph



Dominator Representation

|  |  |  |
|---|---|---|
| 1 | DOM | all the nodes |
| 2 | DOM | 2 |
| 3 | DOM | all the nodes except 1,2 |
| 4 | DOM | all the nodes except 1,2,3 |
| 5 | DOM | 5 |
| 6 | DOM | 6 |
| 7 | DOM | 7,8,9,10 |
| 8 | DOM | 8,9,10 |
| 9 | DOM | 9 |
| 10 | DOM | 10 |

- The dominated information is represented in a tree is called as Dominator tree.



ii) Natural Loops:

- A good 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. Such edges are called back edges.
- 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 the loop.

iii)

iv)Pr

- Natural loop of the edge 10→7 consists of nodes 7,8,10.
- Since 8 and 10 are the nodes that can reach 10 without going through 7.

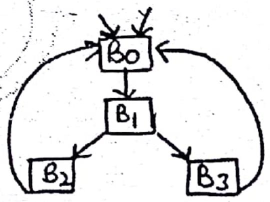
(*) Algorithm for constructing the Natural loops

Input:- A flowgraph GF and a back edge N→D
Output:- A set of loop of all nodes in the Natural loop of N→D

```
Procedure Insert(M)
M is not in loop then
        Loop = {Loop} U {M}
        Push(M)
end
/* Main Program */
 Stack = empty
 Loop = {D}
 Insert(N)
 While (Stack not empty)
   {
      Pop(M)
      for(each predecessor P of M)
         Insert(P)
   }
```

iii) Inner Loops:

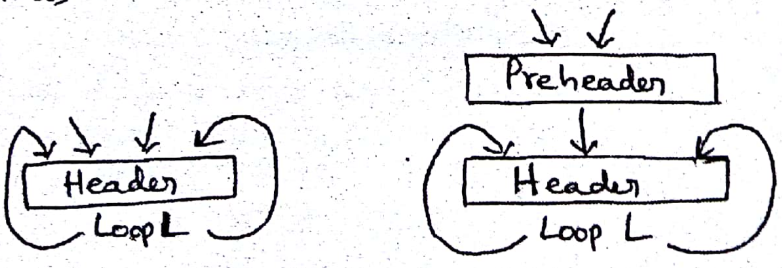
- A loop that contains no other loop is called an inner loop.
- One of the property of the loops is that unless two loops have the same header, they are either disjoint or one is entirely contained within the other(nested).
- When two loops have the same header, it is difficult to tell which is the inner loop.
- Thus, we assume that, when two natural loops have the same header, but neither is nested within the other, they are combined and treated as a single loop.



Two Loops with the same header

iv) Pre-headers:

- Some transformations require the control to move statements before the header.
- Thus, we can begin treatment of a loop L by creating a new block, called the Preheader.
- The Preheader has only the header as successor and all edges which entered the header of L from outside L, now enter the preheader.
- Edges from inside loop L to the header are not changed.
- This is shown as



(a) Before (b) after

Introduction of the preheader

Scanned by CamScanner

**Q22. Explain reducible and non-reducible flow graphs with an example.**

**Answer :**

**Reducible Flow Graphs**

A flow graph $G_F$ is said to be reducible, if its edges can be divided into two disjoint groups i.e., forward edges and back edges and it has two properties. They are as follows,

1. The back edges group contains only those edges whose heads dominates their tails.
2. The forward edges make an acyclic graph, with every node reachable from the initial node of '$G_F$'.

Almost all the flow graphs are reducible. For example, flow graphs for statements like while-do, continue, if-then-else, break and goto are reducible.

In reducible flow graphs, there are no jumps in the middle of the loop from outside. Hence, entry into a loop is through its header.

The example of reducible flow graph is as shown in figure (1).

**Example**

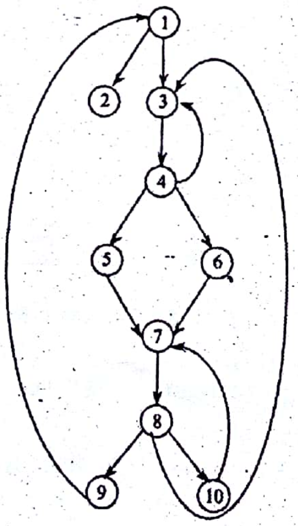


**Figure (1): Reducible Flow Graph**

Scanned by CamScanner

The back edges in a flow graph can be found and eliminated if DOM (Dominance) relation for a flow graph is known. To cross check whether the above flow graph is reducible or not, remove the back edges $4 \rightarrow 3$, $10 \rightarrow 7$, $8 \rightarrow 3$ and $9 \rightarrow 1$. As these are the back edges and the remaining are forward edges, the graph is acyclic (after removal of back edges) and hence reducible.

**Non-reducible Flow Graphs**

There are some flow graphs which are not reducible, such a flow graph is as shown in figure (2).

**Example**

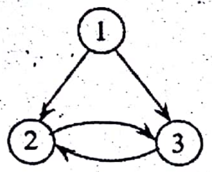


**Figure (2): Non-reducible Flow Graph**

Here, there are no back edges as no head of an edge dominates its tail. As, the cycle $2 \rightarrow 3$ can be entered at two different places, node 2 and node 3, the flow graph is non-reducible. However, it can be reducible if the complete flow graph is acyclic.

Non-reducible flow graphs are very rare. Languages like Modula 2 and Bliss allow programs with reducible flow graphs only.

**Q24. Why next-use information is required for generating object code?**

**Model Paper-III, Q6(b)**

**Answer :**

In a three-address statement, the use of a name is defined as follows. If a statement *i* assigns a value to *x* and there is another statement *j* that uses *x* as an operand and there are no other assignments to *x* between the statements *i* and *j*, then it is said that there is next use of *x* computed at *i* in statement *j*.

In order to determine the next use of a name, first determine the ends of basic blocks by making a backward pass over each basic block. Next we scan each block in the backward direction and determine the next use and liveness of each name *x*. If *x* has next use in the block, then it is recorded in the symbol table otherwise we determine whether the name is live outside this block and record it in the symbol table. From data-flow analysis, we can determine which names are live outside each block. If live-variable analysis is not done, then it is assumed that all non-temporary names are live outside each block. Some temporary names are also considered live across blocks if it is permitted by the code generation algorithm or code optimization algorithm. Such temporaries are marked to distinguish them from non-live temporaries.

In the backward scan of a basic block, when we reach a three-address statement such as $x := y$ op $z$ we do the following.

1. Attach the information found about next use and liveness of *x*, *y* and *z* from symbol table to the to statement *i*.
2. In the symbol table, set the entry for *x* to "not live" and "no next use".
3. In the symbol table, set the entry for *y* and *z* to "live" and their next use to *i*.

The order of steps (2) and (3) is not interchangeable because *x* may be *y* or *z*.

The same steps are performed for three-address statement *i* of the form $x := y$ or $x := op\ y$ by ignoring *z*.

**Applications of Next-Use Information**

Machine instructions involving registers are shorter and faster than the instructions involving operands in memory. Therefore, it is important to utilize registers efficiently. If a name held in register is no longer needed, then the register can be used to assign it to some other name. That is storing a name in register only if it will be used subsequently.

The information about next use of a name has a number of applications.

1. It can be used to assign space for attribute values.
2. It assists in register assignment.
3. It can be used to assign storage for temporary names.

**Reuse of Temporaries**

It is sometimes convenient to create a distinct name for each temporary name during optimization. But, each time a temporary name is created, it must be allocated space to hold temporary value. The amount of space required grows with the number of temporaries. The space can be saved if the temporary names are reused.

From the next-use information we can pack two temporaries into the same location if they are not live simultaneously. The storage for temporaries is allocated by examining each in turn.

A temporary that does not contain a line temporary is assigned to the first location in the field for temporaries. If it is not possible to assign a temporary name to any previously created location, then the storage for it is allocated in the data area for the current procedure. Most often, temporaries are packed into registers rather than memory locations.

In the following example, the six temporaries in the basic block are packed into two locations $t_1$ and $t_2$.

$$\begin{array}{lcl}
t_1 := x * y & & t_1 := x * y \\
t_2 := x + x & & t_2 := x + x \\
t_3 := 5 * t_2 & \Rightarrow & t_2 := 5 * t_2 \\
t_4 := t_1 * t_3 & & t_1 := t_1 * t_2 \\
t_5 := y * y & & t_2 := y * y \\
t_6 := t_4 + t_5 & & t_1 := t_1 + t_2
\end{array}$$

## A Simple Code Generator

**Q25. Explain in brief about simple code generator. Also explain simple code generator algorithm with the function GETREG.**

**Answer :** **Model Paper-II, Q6(b)**

**Simple Code Generator**

A simple code generator algorithm helps in generating code for single basic block. This algorithm uses three address instruction inorder to keep track of values stored in register. This helps in identifying the location of values in their respective register and also for avoiding unnecessary loads and stores operations.

This algorithm explains the efficient use of register based on following four principles.

1. They are used for storing few or all the operands associated with an executing operations.
2. They are used for managing runtime storage which include managing of,
   (i) Run-time stack
   (ii) Stack pointers.
3. They are used for storing global values which belong to a block and is used in another blocks.
4. They are used for storing the result of subexpression (when a larger expression is being computed), there by making themselves a good temporaries.

**Algorithm**

Code generation algorithm uses sequential three-address statements consisting a basic block as input. The algorithm consists of the following steps which are performed for each three-address statement of the form, $x := y \; op \; z$.

1. Call a function GETREG to determine the location *L* for storing result of expression *y op z*. *L* could be a register (usually) or a memory location.

2. Look up the entry for *y* in the address descriptor to determine *y*' (one of) the current location (*s*) of *y*. If *y* is currently. In both the register and the memory location, then take *y*' in register. If *y* is not in *L* then generate machine instruction move *y*', *L* move a copy of *y* in *L*.

3. Generate the machine instruction op *z*', *L* where *z*' indicates the current location of *z*. Again if *z* is in both the register and the memory location, then take *z*' value in register. Update *x*'s address descriptor to indicate that *x* is in *L*. If *L* is in register, then update the register descriptor of *L* to indicate that it contains *x* value. Finally, delete *x* from all other register descriptors.

4. If *y* and *z* are in registers, there are no next uses of these operands and they are not live at the end of the block, then update their register descriptors to indicate that those registers no longer contain the values of *y* and/or *z* after executing the instruction $x := y \; op \; z$.

The same steps are required to generate the code for three-address statements of the form $x := op \; y$.

A three-address statement of the form $x := y$ is a special case. There are two cases to generate code for such statements.

**Case (i) : y is in Register**

If this is the case, we simply update the address and register descriptors to record that *x* value is available in a register containing the value of *y*. If there is no next use of *y* and it is not live at the end of the block, then the register no longer holds *y* value.

**Case (ii) : y is in Memory Location**

If *y* is in memory locations, then we can't record that *x* value is in the location of *y*, since changing the value of *y* can't preserve the value of *x*. Therefore, we use a function GETREG to determine a register to hold the value of *y* in it and then make that register as the location of *y*.

An alternative is to generate the instruction MOV *y*, *x*. This is preferred if there is no next use of *x* in the block.

The GETREG function is defined as given below,

**GETREG Function**

For a three-address statement $x := y \; op \; z$, the function GETREG returns a location *L* to store the value of *x*.

1. If a register holds the value of only one variable i.e., *y* (a copy statement $x := y$ causes a register to hold, the value of more than one variable) and after the execution of $x : y \; op \; z$, *y* is not live and has no next use, then return the location for *L* as the register of *y*.

2. If condition 1 fails, then return an empty register for *L*, if available.

3. If condition 2 fails, then determine whether there is next use of variable *x* in the block or *op* is an operator such that it requires a register (e.g. indexing) then do the following.

   ❖ Find an occupied register *R* such that it is referenced furthest in the future or whose value is also available in the memory.

   ❖ Generate the instruction MOV *R*, *M* to store the value of *R* into a memory location *M* if it is not already in *M*.

   ❖ If the register *R* holds the value of more than one variable, then generate a MOV instruction for each variable in *R* to save its value.

4. If *x* has no next uses or there is no suitable occupied register, then return the memory location of *x* as *L*.

By processing the three-address statements in the basic block the variables that are live are stored on memory location by generating MOV instructions.

If the register descriptor specify the name of variables that are present in registers, then the address descriptor will determine whether the same name is already present in its memory location or not. The live variable information obtained by data-flow analysis among blocks also specify what variables are present on exit so that they can be stored. If no such information is obtained by data flow analysis, then all user-defined variables are considered as live at the end of the block.

**Example**

Consider the three-address code for the assignment statement $r, = (x + y) * (x - z) + (x - z)$

$t_1 : = x + y$

$t_2 : = x - z$

$t_3 : = t_1 * t_2$

$r : = t_3 + t_2$

It is assumed that $a$, $b$ and $c$ are always in memory and the temporaries $t_1$, $t_2$ and $t_3$ are not in memory unless their values are explicitly stored by MOV instructions.

For the above sequence of three-address statements, the code generation algorithm produces the following code.

| Three-address | Generated Code Statement | Contents of Register Descriptor | Content of Address Descriptor |
|---|---|---|---|
| | | All registers are empty | |
| $t_1 := x + y$ | MOV $x$, $R_0$ | $R_0$ holds $t_1$ | $t_1$ is in $R_0$ |
| | ADD $y$, $R_0$ | | |
| $t_2 := y - z$ | MOV $x$, $R_1$ | $R_0$ holds $t_1$ | $t_1$ is in $R_0$ |
| | SUB $z$, $R_1$ | $R_1$ holds $t_2$ | $t_2$ is in $R_1$ |
| $t_1 := t_1 * t_2$ | MUL $R_1$, $R_0$ | $R_0$ holds $t_3$ | $t_3$ is in $R_0$ |
| | | $R_1$ holds $t_2$ | $t_2$ is in $R_1$ |
| $r := t_3 + t_2$ | ADD $R_1$, $R_0$ | $R_0$ holds $r$ | $r$ is in $R_0$ |
| | MOV $R_0$, $r$ | | $r$ is in $R_0$ and memory |

When the GETREG function is called for the location to compute $t_1$ it returns $R_0$ as it is available. Since $x$ value is not in $R_0$ it is loaded into it by generating the instruction MOV $x$, $R_0$ then we generate instruction ADD $y$, $R_0$. Next we update the register descriptor to indicate that $R_0$ holds the value of $t$.

Similarly the code for other statements is generated. After the statement $r = t_3 + t_2$ the register $R_1$ becomes empty because the variable $t_2$ contained in it has no next use. Since the variable $r$ is live at the end of the block, it is stored in memory location by generating the instruction MOV $R_0$, $r$.

**Q26. Describe in detail about a simple code generator with the appropriate algorithm.**

**Answer :**

**Nov.-12, Set-3, Q6**

A simple code generator can be defined as an algorithm which generates code for a single basic block. The task of this algorithm is to consider a three-address instruction and keep a record of values stored in registers in order to avoid useless production of loads and stores.

The code generation algorithm uses register and address descriptors. They help in register allocation.

**Register Descriptor**

A register descriptor keeps the information about each register. It associates with each register a list of variables whose values are held in that register. Initially all registers in a register descriptor are empty. As the process of code generation for a block progresses, each register holds a value of zero or more variables at any instant of time. The information stored in a register descriptor is needed whenever there is a need of a new register.

## Address Descriptor

An address descriptor also known as variable descriptor, keeps information about locations of each variable where the current value of the variable can be found at run time. The locations could be a register, memory address or both because the variable was just loaded from memory into register and its value has not changed. This information can be stored in the symbol table and used to access a name.

With the use of descriptors, one can determine what values are present in registers and reuse them, if needed be. Also reclaim registers when it is known that the values held in registers are no longer required in subsequent code or sending the values back into appropriate memory locations.

**Example**

Consider the basic block and its corresponding dag representation given below.

$l_1 : t_1 = x * y$

$x = t_1$

$t_2 = y - 1$

$y = t_2$

$t_3 = y == 0$

if false $t_2$ goto $l_1$

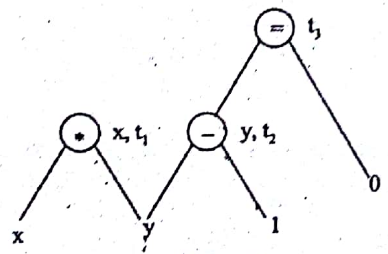


Figure: Dag

Now consider the generation of TM code. Assume that there are three registers $R_0$, $R_1$ and $R_2$ and there are four address descriptors namely inRegister (rno), isGlobal (goffset), isTemp (toffset) and isConst (val). Assume also, that $x$ and $y$ are in global location 0 and 1 respectively. Global location can be accessed through the *gp* register and temporary locations can be accessed through the *mp* register.

Initially all the registers are empty. The address descriptor table begins before code generation for the basic block as given below,

| Variable Name | Address |
|---|---|
| $x$ | isGlobal (1) |
| $y$ | isGlobal (0) |
| $t_1$ | — |
| $t_2$ | — |
| $t_3$ | — |
| 1 | isConst (1) |
| 0 | isConst (0) |

Initially the register descriptor is empty.

Suppose that the following code is generated.

```
LD   0, 1 (gp)     /* load x into R0 */
LD   1, 0 (gp)     /* load y into R1 */
MUL 0, 0, 1      /* Multiply R0 and R1 and store result into R0 */
```

Now, the address descriptor table contents would be,

| Variable Name | Address |
|---|---|
| $x$ | inRegister (0) |
| $y$ | isGlobal (0), inRegister (1) |
| $t_1$ | inRegister (0) |
| $t_2$ | — |
| $t_3$ | — |
| 1 | isConst (1) |
| 0 | isConst (0) |

and the register descriptor is,

| Register | Variables |
|---|---|
| 0 | $x, t_1$ |
| 1 | $y$ |
| 2 | – |

Now the subsequent code generated is,

LDC 2, 1 (0) /* load constant value 1 into $R_2$ */

SUB 1, 1, 2 /* Subtract $R_1$ and $R_2$ and stores result into $R_1$ */

Now, the address descriptor table given as,

| Variable Name | Address |
|---|---|
| $x$ | inRegister (0) |
| $y$ | inRegister (1) |
| $t_1$ | inRegister (0) |
| $t_2$ | inRegister (1) |
| $t_3$ | — |
| 1 | isConst (1), inRegister (2) |
| 0 | isConst (0) |

and the register descriptor is given as follows,

| Register | Variables |
|---|---|
| 0 | $x, t_1$ |
| 1 | $y, t_2$ |
| 2 | 1 |

## SHORT QUESTIONS WITH SOLUTIONS

**Q1. What is an activation tree?**

**Answer :**

**Model Paper-I, Q1(a)**

An activation tree is a tree that shows how control enters and leaves activations.

In an activation tree,

(i) Every node corresponds to activation of a procedure

(ii) The root node corresponds to the activation of main program.

(iii) The node for activation of procedure '*p*' is said to be parent of node for activation of procedure '*q*' iff the flow of control sequence is from '*p*' to '*q*'.

(iv) The node for activation of procedure '*p*' is said to be the left of the node for activation of procedure '*q*' iff the lifetime of '*p*' occurs earlier than the lifetime of '*q*'.

**Q2. What is an activation record?**

**Answer :**

Activation record or frame is a contiguous block of storage that contains all the information required for a single execution of a procedure. An activation record consists of seven fields as shown in the figure.

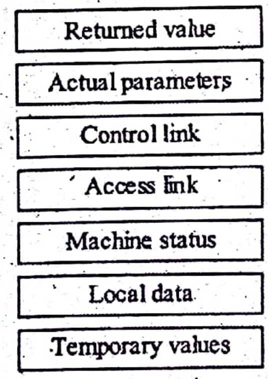


**Figure: An Activation Record**

**Q3. Describe about access links and displays.**

**Answer :**

**Access Links**

In this method a pointer called an access link is added to each activation record. If procedure p1 is nested immediately within p2 then the access link in p1's activation records points to the access link in p2's activation record which is the most recent activation of p2.

**Displays**

This method maintains an array *d* of pointers to activation records called a *display*. An array element *d*[*i*] points to the activation containing the storage for nonlocal *x* with nesting depth *i*.

CD5.7

**Q4. List the basic functions and properties of memory manager.**

**Answer :**

**Functions of Memory Manager**

The two basic functions of memory manager includes,

(i) Memory allocation

(ii) Memory deallocation.

**Properties of Memory Manager**

The properties of memory manager include,

(i) Maintainence of space efficiency

(ii) Maintenance of program efficiency

(iii) Minimizing over head.

**Q5. Write a short note on code generation and list the generic issues in the design of code generator.**

**Answer :** Model Paper-II, Q1(e)

**Code Generation**

Code generation phase is the last phase of compiler. It is an important phase of the compilation because, in this phase the compiler convert the intermediate code into machine or assembly code. Moreover, the allocation of memory to variable in source program is also done by this phase. The code generator assigns register to variable in the program.

**Design Issues of Code Generator**

(i) Input

(ii) Output

(iii) Memory management

(iv) Selection of machine instruction

(v) Allocating register

(vi) Evaluation order

(vii) Code generation.

**Q6. Define Basic block.**

**Answer :**

A basic block is bunch of three-address statements, such that the flow of control if once enters a block of statements then flow continuously without stopping or pausing and ends at the end of the block.

Every statement in a basic block is a three-address statement.

In an assignment statement,

$a := b * c$

$a$ is defined by using $b$ and $c$.

Thus, we can say that the above statement defines '$a$' and uses '$b$' and '$c$'.

A symbol or a variable or a name is known as a live symbol or name if its value is used in future or from that point. Similarly, a name is known as a dead symbol, if its value is not going to be used in the rest of the program.

**Q7. What is flow graph?**

**Answer :** Model Paper-III, Q1(e)

A flow graph is a graphical representation of three-address statements. It is used to show the flow of control information to the set of basic blocks of a program. It consists of nodes and edges. A node of a flow graph is a basic block that performs some computations. An initial node consists of a block whose leader is the first statement. There is a directed edge from one node to another representing the flow of control between blocks. There is an edge from block B1 to block B2.

**Q8. Discuss briefly about natural loops and inner loops of flow graph.**

**Answer :**

**Natural Loops**

In a flow graph, if there exist a back edge $n \rightarrow b$, then the natural loop of the edge '$b$' is given along with a set of nodes that do not go through '$b$' to reach '$n$'. In the edge $n \rightarrow b$, $b$ is the head and $n$ is the tail.

**Inner Loops**

A loop that does contain any other loop is called an inner loop. For example, the flow graph given in the below figure (1), has an inner loop $4 \rightarrow 2$ i.e., the path from 2-3-4.

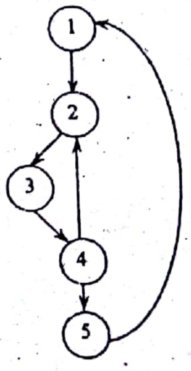


Figure: Flow Graph with an Inner Loop $4 \rightarrow 2$

**Q9. What are reducible and non-reducible flow graphs?**

**Answer :** Model Paper-IV, Q1(e)

**Reducible Flow Graphs**

A flow graph $G_F$ is said to be reducible, if its edges can be divided into two disjoint groups i.e., forward edges and back edges and it has two properties. They are as follows,

1. The back edges group contains only those edges whose heads dominates their tails.

2. The forward edges make an acyclic graph, with every node reachable from the initial node of '$G_F$'.

Scanned by CamScanner

Almost all the flow graphs are reducible. For example, flow graphs for statements like while-do, continue, if-then-else, break and goto are reducible. In reducible flow graphs, there are no jumps in the middle of the loop from outside. Hence, entry into a loop is through its header.

**Example**

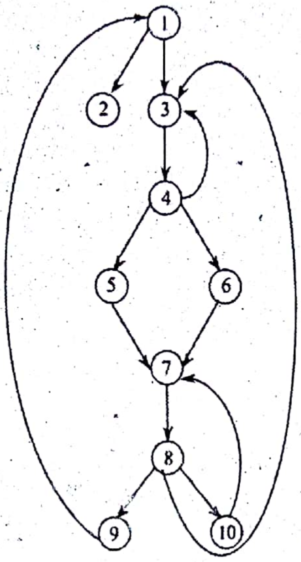


**Figure: Reducible Flow Graph**

**Non-reducible Flow Graphs**

There are some flow graphs which are not reducible, such a flow graph is as shown in the below figure.

**Example**

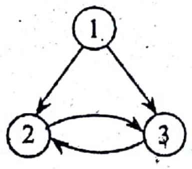


**Figure: Non-reducible Flow Graph**

**Q10. What are the applications of Next-use information?**

**Answer :**

Machine instructions involving registers are shorter and faster than the instructions involving operands in memory. Therefore, it is important to utilize registers efficiently. If a name held in register is no longer needed, then the register can be used to assign it to some other name. That is storing a name in register only if it will be used subsequently.

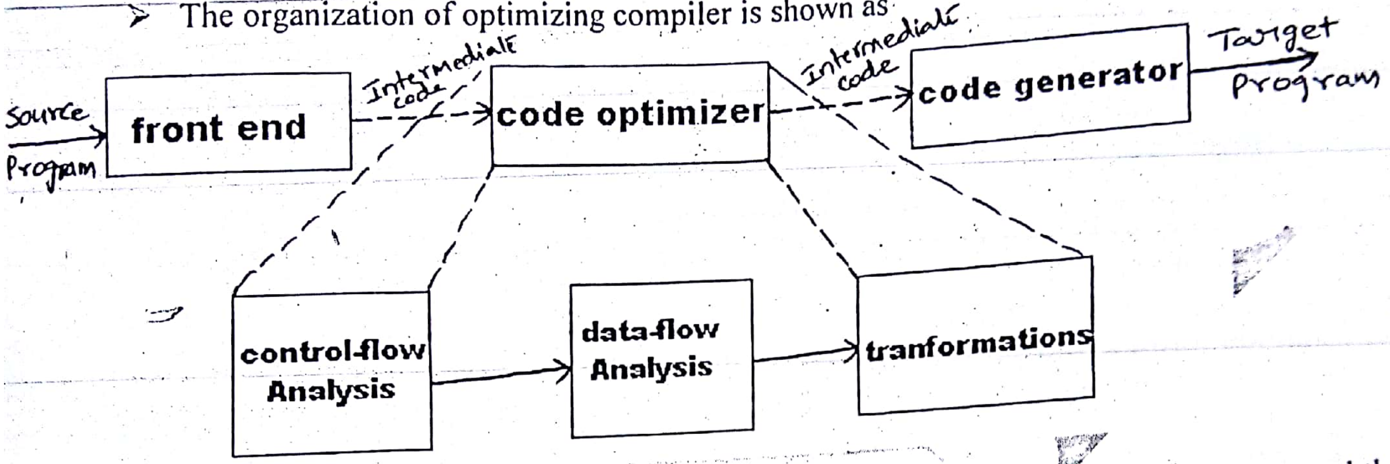
The information about next use of a name has a number of applications.

1. It can be used to assign space for attribute values.
2. It assists in register assignment.
3. It can be used to assign storage for temporary names.

# CD-UNIT-6

## (*)INTRODUCTION TO CODE OPTIMIZATION:

- The code optimization phase attempts to improve the intermediate code , so that it will result faster-running machine code.
- Compilers that apply code-improving transformations are called *Optimizing Compilers.*
- Optimizing a program is required, so that the compiler can generate
  - Efficient target code along with preserving the meaning of the program.
  - Reduction in size of generated code.
- The organization of optimizing compiler is shown as



- The code optimization phase consists of control-flow analysis, data-flow analysis and the application of transformations.
- The code generator produces the target program from the transformed intermediate code.
- The input and output of code optimizer is an Intermediate code.
- There can be situations where a piece of code cannot be optimized. Hence the output from the code optimizer is the same set of intermediate code.

**Advantages of Optimizing Compiler:**

- The operations needed to implement high-level constructs are made explicit in the intermediate code, so it is possible to optimize them.

- As the machine independent phases are separated from machine dependent phase (code generator), the code optimizer can easily optimize code with few changes even if the code generator is for any kind of machine.

**Machine Dependent Optimization:**

- Machine dependent optimization stem from special machine properties that can be exploited to reduce the amount of code or execution time.

*Example:*
Register allocation, data intermixed with instructions and special machine features, exi-vector operation.

**Machine Independent Optimization:**

- Machine Independent Optimization depends on only the arithmetic properties of the operations in the language and not on peculiarities of the target machine.

(1) Machine Independent code optimization:
→ It depends on only arithmetic properties of the operations in the language

## THE PRINCIPLE SOURCES OF OPTIMIZATION:

- 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 global.
- Many transformations can be performed at both the local and global values.
- Local transformations are usually performed first.
- There are number of ways in which a compiler can improve a program without changing the function it computes. They are,
  - a) Function preserving transformations.
  - b) Loop optimizations.

## a). FUNCTION PRESERVING TRANSFORMATIONS: (local Optimization)

- It is also called as local optimization.
- It includes,
  - a) Common Sub-expression Elimination.
  - b) Copy Propagation.
  - c) Dead-Code Elimination.
  - d) Constant Folding.

### a) Common Sub-expression Elimination:

- The code optimization can be improved by eliminating common subexpressions from the code.
- An expression whose value was previously computed and the values of variables in the expression are not changed.
- Since its computation can be avoided to recomputed it by using the earlier computed value.

*Example:*

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

-In this code b*c is common for both 'a' and 'z'

-Then this code is replaced by the code, while eliminating b*c subexpression is,

```
t1:=b*c
a:=t1
z:=t1+d-c
```

- It is not possible to eliminate an expression if the value of one of its variable is changed.

*Example:*

```
a:=b*c
c:=4+c
z:=b*c+d-c
```

-In this code b*c is not common for both 'a' and 'z' because 'c' is changed after a:=b*c is computed.

### b) Copy Propagation:

- It is also called as Variable Propagation.
- If there are copy statements of the form x:=y then in Copy propagation, the use of variable x is replaced by the variable y in the subsequent expression.
- The copy propagation is possible if none of the variable is changed after this arrangement.

*Example:*

```
a:=b+c
d:=c
e:=b+d-3
```

-This code has common subexpressions b+c and b+d.
-If we replace the variable 'd' by the variable 'c' as both have the same value.
-After applying the copy propagation the code is,

```
a:=b+c
d:=c
e:=b+c-3
```

**c) Dead-Code Elimination:**

- A piece of code which is not reachable, that is, the values it computes is never used anywhere in the program then it is said to be dead code.
- It can be removed for program safely.
- An assignment to a variable results in dead code, if the value of this variable is not used in the subsequent program.
- Copy propagation often makes the copy statements into dead code.
  *Example:*

```
a:=b+c
d:=c
e:=b+d-3
```

-Here, let us suppose the value of 'd' and 'a' are not used in the subsequent program, then we eliminate the dead code variable d and a, the code is changed after the elimination of dead code is,

```
t1:=b+c
e:=t1-3
```

**d) Constant Folding:**

- The substitution of values for names whose values are constant is known as Constant Folding.
- Constant folding is simply applied in such a way that values known at compile time to be associated with variables are used to replace certain uses of these variables in the translated program text.
  *Example:*

```
#define PI 3.14
area=PI*r*r;
```

-Here area=PI*r*r; is replaced by area=3.14*r*r; at the time of compilation.

## b) LOOP OPTIMIZATION:

- The major source of code optimization is loops, especially inner loops.
- Most of the run-time is spent inside the loops which can be reduced by reducing the number of instructions in an inner loop.
- It includes,
  - a) Code Motion
  - b) Induction Variable Elimination
  - c) Reduction in Strength

**a) Code Motion :**

➢ Code Motion reduces the number of instructions in a loop by moving instructions outside a loop.
➢ It moves loop-invariant computations.
➢ Loop-invariant computations or expressions that results in the same value independent of the number of times a loop is executed.

*Example:*

```
while(x!=n-2)
  {
    x=x+2;
  }
```

-Here, expression n-2 is a loop-invariant computation.

➢ The expression is changed to

```
m=n-2;
while(x!=m)
  {
    x=x+2;
  }
```

**b) Induction Variable Elimination:**

➢ An induction variable is a loop control variable or any other variable that depends on the induction variable in some fixed way.
➢ It can also be defined as a variable which is incremented or decremented by a fixed number in a loop each time, the loop is executed.
➢ Induction variables are of the form i=i±c, where 'c' is a constant.

*Example:*

```
int a[10],b[10];
void fun(void)
  {
    int i, j, k;
    for(i=0, j=0 ,k=0; i<10; i++)
            a[j++]=b[k++];
    return;
  }
```

-Here,we have three induction variables i, j, k.
-j and k are not used properly, so we eliminate induction variables j and k, then the change is,

```
int a[10], b[10];
void fun(void)
  {
    int i;
    for(i=0;i<10;i++)
          a[i]=b[i];
    return;
  }
```

➢ This property will reduce the code and improves the run-time performance of a compiler.

**c) Reduction in Strength:**

- Strength Reduction is the process of replacing expensive operations by equivalent cheaper operations on the target machine.
- On many machines a multiplication operation takes more time than addition or subtraction.
- On such machines the speed of the object code can be increased by replacing a multiplication by an addition or subtraction. This is called Reduction in Strength.

*Example:*

```
for(i=1;i<5;i++)
    {
      x=4*i;
    }
```

-Here, the instruction x=4*i is equal to x=x+4, so it is replaced by

```
for(i=1;i<5;i++)
    {
      x=x+4;  //where x=0
    }
```

## ② Peephole Optimization:

→ A peephole optimization technique is used to improve the target code.
→ Peephole optimization works by finding the peepholes (a short sequence of target Instructions) and replacing them by a shorter (or) faster sequence of Instructions.
→ It can be applied to the intermediate code (or) the object code.
→ It uses the following transformations to improve the code.

a) Elimination of Redundant instructions
b) Optimization of flow of control
c) Simplification of Algebraic expressions
d) Instructions selection.
(e) Eliminate Dead code
(f) Strength reduction

### a) Elimination of Redundant Instructions,

→ One approach to improve the target code is to remove redundant instructions.

Example,

```
(1) MOV  R1, A
(2) MOV  A, R1
```

→ Here, the first instruction is storing the value of A into register R1 and second instruction storing R1 value into A.
→ These two instructions are redundant, so we eliminate instruction (2)
→ To perform such a transformation, both instructions must be in the same basic block.

(b) Optimize flow of controls:

→ The target code generated by the code generation algorithm frequently contains unnecessary Jumps such as,

- Jumps to Jumps
- Jumps to Conditional Jumps
- Conditional Jumps to Jumps.

Example: Jump to Jump

```
goto label1
    ____
label1: goto label2
```

can be achieved by the sequence

```
goto label2
    ____
label1: goto label2
    ____
  label2:
     ____
```

where, label1: goto label2 is removed, using peephole optimization

(c) Simplify Algebraic expressions:

→ There are endless algebraic simplifications that can be achieved through peephole optimization.

→ One of this is to implement algebraic identities that occur frequently in the code.

→ Examples are,

$a := 0 + a$ (or)

$a := 1 * a$

which can be eliminated easily through peephole optimization.

(d) Instruction Selection:

→ Also called as use of machine idioms.

→ Target machines have hardware instructions that can perform certain operations more efficiently

→ The use of these instructions in the target code significantly reduces the running time of the target program.

→ The addressing modes such as auto-decrement available in some machine causes an operand to be incremented (or) decremented by one automatically.

→ The use of these modes in parameter passing and in pushing (or) popping a stack greatly improves the target code.

→ The statements like,

$x = x + 1$ (or)

$x = x - 1$

Can also be replaced by these addressing modes.

(e) Eliminate Dead code:

→ Another possibility to improve code is to remove unreachable instructions from the target program

→ An unlabeled instruction immediately after the unconditional Jump is unreachable, So it can be removed.

Example:

```
If FLAG=1 goto L1 goto L2

L2:
   ===
```

→ peephole optimization replace the above code as,

```
If FLAG ≠ 1 goto L2
L2:
   ===
```

f) Strength reduction:

→ In strength reduction expensive operations replaced by equivalent cheaper operations on the target machine.

Example: $x^2$ is expensive because it needs a call to an exponentiation routine.

→ So it is cheaper to implement it as a multiplication expression, i.e,

$\boxed{x * x}$

→ It is cheaper to implement,

- a fixed-point multiplication (or) division by a power of two as a shift operation and
- floating-point division by a constant as multiplication by a constant.

## 3. INTRODUCTION TO DATA FLOW ANALYSIS:

- The code improvement phase consists of control-flow analysis, data-flow analysis and application of transformations.

### 1.1 Basic Blocks:

- A Basic Block is a collection of three-address statements in which flow of control enters at the beginning and leaves at the end without halt or possibility of branching except at the end.
- Every statement in a basic block is a three-address statement.

*Example:*

a=(b*c)/d+k

The three-address statements are,

```
t1:=b*c
t2:=t1/d
a:=t2+k
```

### 1.2 Algorithm for Partitioning into Basic Blocks:

- This algorithm is used to partition a sequence of three address statements into basic blocks.

**Input:** A sequence of three-address statements.

**Output:** A list of basic blocks with each three-address statement in exactly one block.

**Method:**

- We first determine the set of leaders, i.e, the first statements of basic blocks.
- The rules used are,
  i)The first statement is a leader.
  ii)Any statement that is the target of a conditional or unconditional goto is a leader.
  iii)Any statement that immediately follows a goto or conditional goto statement is a leader.
- A Basic Block is drawn for each leader followed by the set of statements.
- No Basic Block can have more than one leader.

## ❀ GLOBAL OPTIMIZATION :-

→ The local optimization has a very restricted scope on the other hand the global optimization is applied over a broad scope such as procedure (or) function body.

→ For a global optimization, a program is represented in the form of program flowgraph.

→ The program flowgraph is a graphical representation in which each node represents the basic block and edges represent the flow of control from one block to another.

→ There are two types of analysis performed for global optimizations.

(i) Control-flow Analysis.
(ii) Data-flow Analysis.

→ The control-flow analysis determines the information regarding arrangement of graph nodes (basic blocks), presence of loops, nesting of loops and so.. on.

→ In control-flow analysis, the analysis is made on the flow of control by carefully examining the program flowgraph.

→ In Data-flow Analysis, the analysis is made on flow of data.

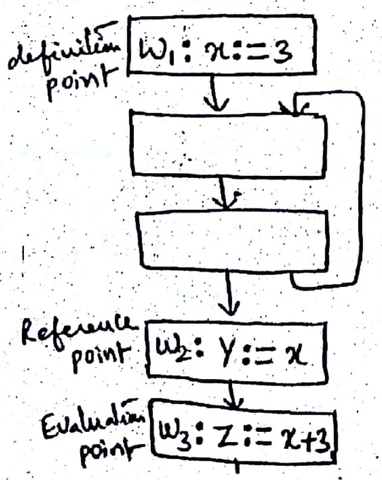
→ The Data-flow Analysis is basically a process in which the values are computed using data flow properties such as

- Available expressions
- Reaching definitions
- Live Variables
- Busy ~~Ex~~ Expressions.

→ The basic terminologies on data flow properties are,

- A program point containing the definition is called definition point.
- A Program point at which a reference to a data item is made is called reference point.
- A Program point at which some evaluating expression is given is called Evaluation point.

Ex:-



Program Point (w)

Scanned by CamScanner

## (*) Data-Flow-Properties :-

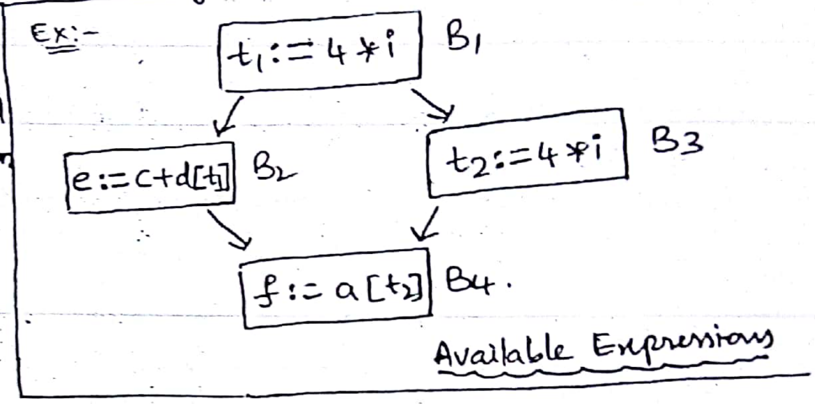
(i) Available Expressions
(ii) Reaching Definitions
(iii) Live Variables
(iv) Busy Expressions

(i) Available Expressions :-

→ An expression is available at a program point (w) if and only if along all paths are reaching to w.

→ The expression 4*i is the available for $B_2$, $B_3$ and $B_4$, because this expression is not been changed by any of the block before appearing in $B_4$.

→ The use of available expressions is to eliminate common subexpressions.

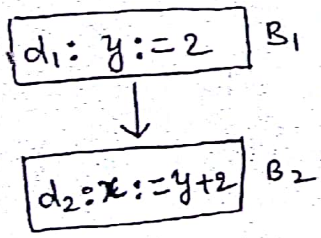
Ex:-


Available Expressions

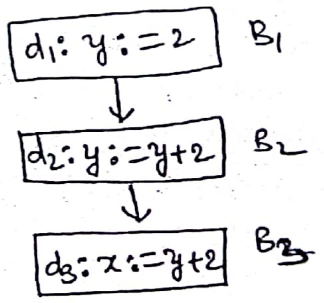
(ii) Reaching Definitions :-

→ A definition 'd' reaches at point 'p' if there is a path from 'd' to 'p' along which 'd' is not killed.

→ A definition 'd' of a variable 'x' is killed when there is a redefinition of 'x'.

Ex:-


Reaching definition $d_1$



Killing definition $d_1$

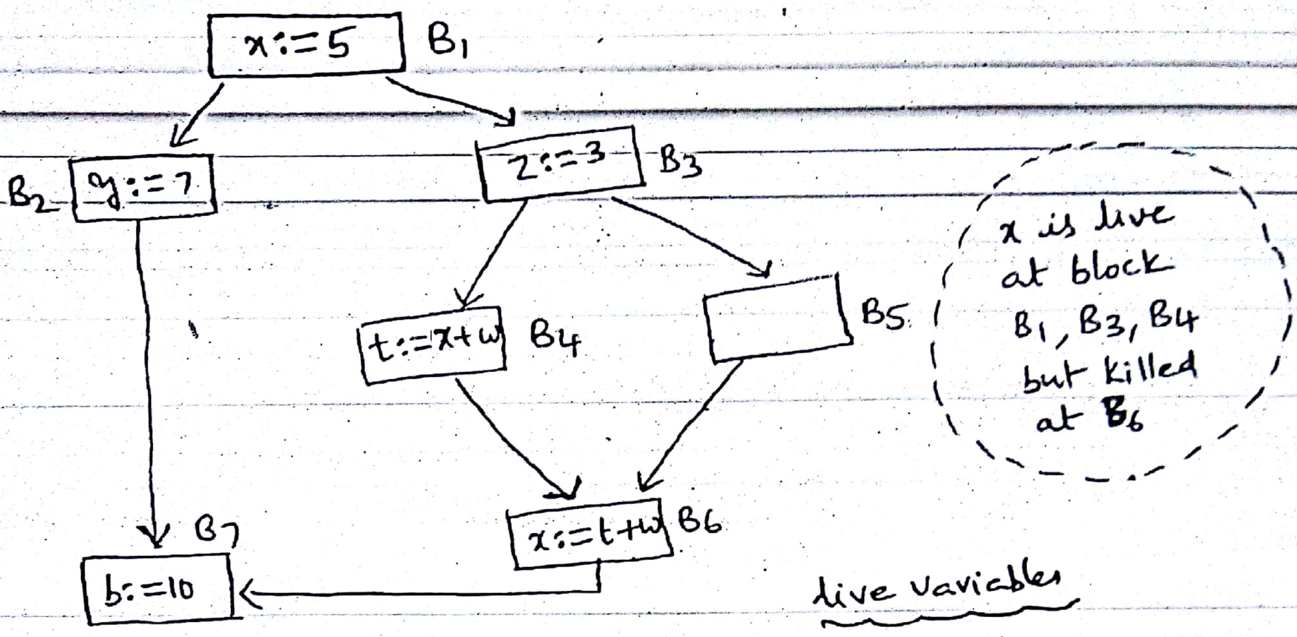
→ Reaching definitions are used in Constant and variable propagation.

(iii) Live Variables :-

→ A variable 'x' is live at some point 'p' if there is a path from 'p' to the exit, along which the value of x is used before it is redefined.

→ Otherwise the variable is said to be dead at that point.

Ex:-



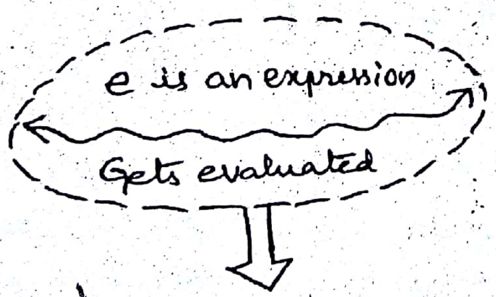
live variables

→ Live variables are useful in
- register allocation
- Dead code elimination.

(iv) Busy Expressions :-

→ An expression 'e' is said to be a busy expression along some path $P_i$ ------ $P_j$ if and only if an evaluation of e exists along some path $P_i$ ----- $P_j$

→ No definition of any operand exists before its evaluation along the path.



e = Busy Expression

Busy Expressions

→ Busy Expressions are useful in performing code movement optimization.

## Q15. Write an algorithm to compute reaching definition informatory for a flow graph.

**Answer :**

A statement that assigns a value to a variable *V* is a definition of *V*. A definition *d* of a variable *V* is said to reach a point *p*, if there exists a path from *d* to *p* along which *V* is not redefined. The reaching definition problem is to compute for each block, all definitions of each variable which reach the beginning of the block.

The data flow equations for reaching definitions are as follows,

$$BGN[B] = \bigcup_{\substack{P \text{ is a} \\ \text{Predecessor} \\ \text{of } B}} END[P]$$

$$END[B] = GEN[B] \cup (BGN[B] - KILL[B])$$

The union operator ($\cup$) in equation for BGN[B] indicates that as definition reaches a block if it reaches the end of any of its predecessors.

**Algorithm**

The algorithm for computing reaching definitions takes a flow graph as input for which the sets GEN[*B*] and KILL[*B*] have been computed for each block B. The output of the algorithm is the sets BGN[B] and END[B] computed for each block *B*.

The algorithm initially assumes that BGN[B] = $\phi$ for each block B. It propagates reaching definitions information as long as they are not killed the algorithm is as follows.

1. Initialize END[B] := GEN[B] for each block B on the assumption that initially BGN[B] = $\phi$ for all B.
2. Do following until there are no changes in any of the END[B] sets.
   (i) For each block *B* calculate

   $$BGN[B] := \bigcup_{\substack{P \text{ is a} \\ \text{Predecessor} \\ \text{of } B}} END[P]$$

   PREV_END[B] := END[B]

   END[B] := GEN[B] $\cup$ (BGN[B] – KILL[B]);

   (ii) Check whether there are any changes in END[B] sets by comparing PREV_END[B] and END[B]. Record the result this helps in terminating the algorithm.

## Q16. Explain the working of the above algorithm using a suitable example.

**Answer :**

Model Paper-III, Q7(b)

Consider the graph shown below,

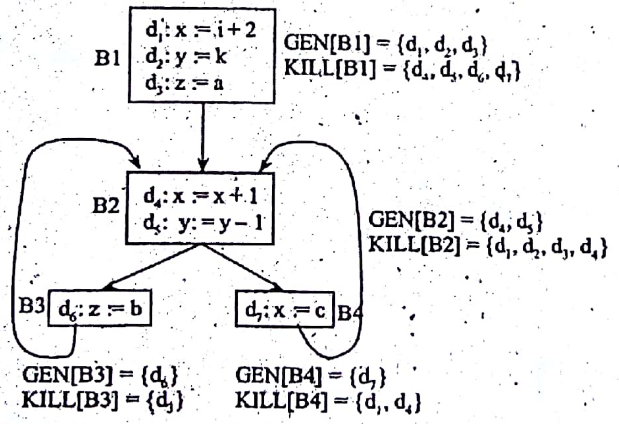


**Figure: Flow Graph**

The above figure shows GEN and KILL sets computed for each block. In the above flow graph there are seven definitions $d_1, d_2, ..., d_7$ defining the variables *x*, *y* and *z*.

We use bit vectors to represent the set of definitions. In a bit vector, the bit *i* contains 1 if and only if the definition $d_i$ is in the set.

The first step of the algorithm initializes END[B] = GEN[B] for each block B on the assumption that initially BGN[B] = $\phi$ for all B. These initial values of END[B] are shown in the table given below.

Scanned by CamScanner

CD 6.8

The algorithm enters into the while loop and starts first iteration. Suppose in the inner for loop B takes B1, B2, B3 and B4 in that order with B = B1.

BGN[B1] = $\phi$ represented by 000 0000 since B1 is an initial node which have no predecessors and

END[B1] = GEN[B1]

Which also remains equal to GEN[B1] since PREV_END[B1] = END[B1] the variable any changes is not set to true.

Next for loop takes B = B2 for which BGN and END sets are computed as follows,

BGN[B2] = END[B1] $\cup$ END[B3] $\cup$ END[B4]

= 1110000 + 0000 010 + 0000 001

= 1110 011

END[B2] = GEN[B2] $\cup$ (BGN[B2] – KILL[B2])

= 0001 100 + (1110 011 – 1100 001)

= 0011 110

Similarly BGN and END sets for B3 and B4 are computed the table given below shows these computations.

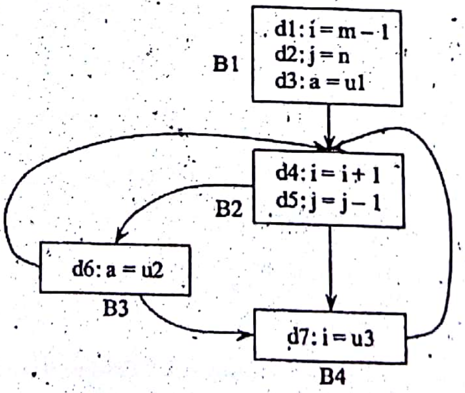
| Block | Initial Values | | Iteration 1 | | Iteration 2 | |
|---|---|---|---|---|---|---|
| B | BGN[B] | END[B] | BGN[B] | END[B] | BGN[B] | END[B] |
| B1 | 0000000 | 1110000 | 0000000 | 1110000 | 0000000 | 1110000 |
| B2 | 0000000 | 0001100 | 1110011 | 0011110 | 1111111 | 0011110 |
| B3 | 0000000 | 0000010 | 0011110 | 0001110 | 0011110 | 0001110 |
| B4 | 0000000 | 0000001 | 0011110 | 0010111 | 0011110 | 0010111 |

**Table: Computation of BGN and END Sets for Example Flow Graph**

At the end of the first iteration the set END[B4] = 0010111 represents that the definition $d_7$ is generated in B4 and definitions $d_3$, $d_5$ and $d_6$ reach B4 without being killed in B4.

The second iteration of while loop starts. In this iteration there are no changes to any of the END sets so the algorithm terminates.

**Q17. Write the iterative algorithm for reaching definition. Compute in and out for the following figure.**

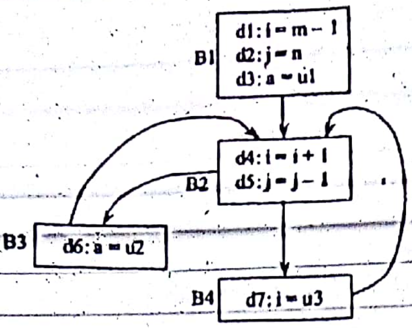


**Figure**

**Answer :**

**Model Paper-II, Q7(b)**

**Iterative Algorithm for Reaching Definition**



**Figure**

In the figure of flow graph, there are seven definitions $d_1, d_2 \ldots\ldots d_7$ defining the variables $i, j$ and $a$.

The GEN and KILL sets for each block are as follows,

For B1: GEN[B1] = $\{d_1, d_2, d_3\}$

KILL[B1] = $\{d_4, d_5, d_6, d_7\}$

For B2: GEN[B2] = $\{d_4, d_5\}$

KILL[B2] = $\{d_1, d_2, d_7\}$

For B3: GEN[B3] = $\{d_6\}$

KILL[B3] = $\{d_3\}$

For B4: GEN[B4] = $\{d_7\}$

KILL[B4] = $\{d_1, d_4\}$

We use bit vectors to represent the set of definitions. In a bit vector, the bit $i$ contains 1 if and only if the definition $d_i$ is in the set.

The first step of the algorithm initializes END[B] = GEN[B] for each block B on the assumption that initially BGN[B] = $\phi$ for all B. These initial values of END[B] are shown in the table.

**First Iteration**

The algorithm enters into the while loop and *starts first iteration*. Suppose in the inner for loop B takes B1, B2, B3 and B4 in that order with B = B1.

BGN[B1] = $\phi$ represented by 000 0000 since B1 is an initial node which have no predecessors and

END[B1] = GEN[B1] = 1110000

Which also remains equal to GEN[B1] since PREV_END[B1]=END[B1] the variable any changes is not set to true.

Next, for loop takes B = B2 for which BGN and END sets are computed as follows,

BGN[B2] = END[B1] ∪ END[B3] ∪ END[B4]

= GEN[B1] ∪ 0000000 ∪ 0000000

= 1110000 + 0000000 + 0000000

= 1110000

END[B2] = GEN[B2] ∪ (BGN[B2] − KILL[B2])

= 0001100 + (1110000 − 1100001)

= 0001100 + 0010000

= 0011100

Now, for loop takes B = B3 for which BGN and END sets are computed as follows,

BGN[B3] = END[B2]

= 0011100

END[B3] = GEN[B3] ∪ (BGN[B3] – KILL[B3])

= 0000010 + (0011100 – 0010000)

= 0000010 + 0001100

= 0001110

Now, for loop takes B = B4 for which BGN and END sets are computed as follows,

BGN[B4] = END[B2]

= 0011100

END[B4] = GEN[B4] ∪ (BGN[B4] – KILL[B4])

= 0000001 + (0011100 – 1001000)

= 0000001 + 0010100

= 0010101

**Second Iteration**

The algorithm enters into the while loop again and *starts second iteration*. Here, we consider the BGN and END sets of each block obtained in first iteration.

For B = B1; BGN[B1] = ϕ = 0000000

END[B1] = GEN[B1]

= 1110000

For B = B2; BGN[B2] = END[B1] ∪ END[B3] ∪ END[B4]

= 1110000 + 0001110 + 0010101

= 1111111

END[B2] = GEN[B2] ∪ (BGN [B2] – KILL[B2])

= 0001100 + (1111111 – 1100001)

= 0001100 + 0011110

= 0011110

For B = B3; BGN[B3] = END[B2]

= 0011110

END[B3] = GEN[B3] ∪ (BGN[B3] – KILL[B3])

= 0000010 + (0011110 – 0010000)

= 0000010 + 0001110

= 0001110

For B = B4; BGN[B4]= END[B2]

= 0011110

END[B4]= GEN[B4] ∪ (BGN[B4] – KILL[B4])

= 0000001 + (0011110 – 1001000)

= 0000001 + (0010110)

= 0010111

**hird Iteration**

The algorithm enters into the while loop again and starts third iteration. Here, we consider the BGN and END sets of each lock obtained in second iteration.

For B = B1; BGN[B1]= ϕ = 0000000

END[B1]= GEN[B1]

= 1110000

For B = B2; BGN[B2]= END[B1] ∪ END[B3] ∪ END[B4]

= 1110000 + 0001110 + 0010111

= 1111111

END[B2]= GEN[B2] ∪ (BGN[B2] – KILL[B2])

= 0001100 + (1111111 – 1100001)

= 0001100 + 0011110

= 0011110

For B = B3; BGN[B3]= END[B2]

= 0011110

END[B3]= GEN[B3] ∪ (BGN[B3] – KILL[B3])

= 0000010 + (0011110 – 0010000)

= 0001110

For B = B4; BGN[B4]= END[B2]

= 0011110

END[B4]= GEN[B4] ∪ (BGN[B4] – KILL[B4])

= 000001 + (0011110 – 1001000)

= 000001 + 0010110

= 0010111

In the third iteration of while loop there are no changes to any of the END sets. So, the algorithm terminates.

The table shows all the BGN and END sets for the given flow graph:

| Block | Initial Values | | Iteration 1 | | Iteration 2 | |
|---|---|---|---|---|---|---|
| B | BGN[B] | END[B] | BGN[B] | END[B] | BGN[B] | END[B] |
| B1 | 0000000 | 1110000 | 0000000 | 1110000 | 0000000 | 1110000 |
| B2 | 0000000 | 0001100 | 1110011 | 0011110 | 1111111 | 0011110 |
| B3 | 0000000 | 0000010 | 0011110 | 0001110 | 0011110 | 0001110 |
| B4 | 0000000 | 0000001 | 0011110 | 0010111 | 0011110 | 0010111 |

## SHORT QUESTIONS WITH SOLUTIONS

**Q1. Discuss briefly about,**

**(i) Local optimizations**

**(ii) Global optimizations.**

**Answer :** Model Paper-I, Q1(f)

(i) **Local Optimizations**

These are the optimizations carried out within a single basic block. This technique do not require the information regarding the data and flow of control. Thus, implementation of this technique is simple.

(ii) **Global Optimizations**

These are the optimizations carried out across basic blocks, instead of single basic block. This analysis is also known as data-flow analysis. In this technique, additional analysis is required across basic blocks. Thus, implementation of this technique is complex.

**Q2. Briefly explain how global common sub expressions are eliminated.**

**Answer :**

**Eliminating Global Common Subexpression**

The code can be improved by eliminating common subexpressions from the code. An expression whose value was previously computed and the values of variables in the expression are not changed, since its computation can be avoided to recompute it by using the earlier computed value.

**Example**

Consider the following sequence of code.

```
a := b * c
  ⋮
z := b * c + d – c
```

In the above code the assignment to *z* have the common subexpression *b* * *c*. Since its value is not changed after the point it was computed, and its use in the expression *z*, we can avoid recomputing it by replacing the above code as follows:

$t_1 := b * c$

$a := t_1$

⋮

$z := t_1 + d - c$

However, it is not possible to eliminate an expression if the value of one of its variable is changed.

Consider the following code:

```
a := b * c
c := 4 + c
  :
z := b * c + d - c
```

Here, the expression $b * c$ is not common because the value of variable $c$ is changed after computing $b * c$. Therefore, we cannot eliminate this expression.

**Q3. Explain briefly elimination of redundant instruction in reephole optimization.**

**Answer :**

One approach to improve the target code is to remove redundant instructions. For example, consider the following instructions.

1. MOV R1, A
2. MOV A, R1.

Here the first instruction is storing the value of $A$ into register R1 and second instruction is loading R1 value into $A$. These two instructions are redundant so we eliminate instruction (2), because whenever instruction (2), is executed after (1), it is ensured that the register R1 contains $A$ value. But of the instruction (2), had a label then we could not delete it because we are not sure that (1) will always be executed before (2). To perform such a transformation both instructions must be in the same basic block.

**Q4. Discuss briefly about,**

**(i) Simplification of algebraic expressions**

**(ii) The use of machine instructions.**

**Answer :**

**(i) Simplification of Algebraic Expressions** **Model Paper-IV, Q1(f)**

There are endless algebraic simplifications that can be achieved through peephole optimization. One of this is to implement algebraic identities that occur frequently in the code. For example, the intermediate code generator often produces statements such as,

$$a := 0 + a \text{ or } a = 1 * a$$

Which can be eliminated easily through peephole optimization.

**(ii) The Use of Machine Instructions**

In strength reduction expensive operations replaced by equivalent cheaper operations on the target machine. For example, the expression $x^2$ is expensive because it needs a call to an exponentiation routine. So it is cheaper to implement it as a multiplication expression i.e., $x * x$. It is cheaper to implement a fixed-point multiplication or division by a power of two as a shift operation and floating-point division by a constant as multiplication by a constant.

Target machines have hardware instructions that can perform certain operations more efficiently. The use of these instructions in the target code significantly reduces the running time of the target program. The addressing modes such as auto-decrement available in some machine causes an operand to be incremented or decremented by one automatically. The use of these modes in parameter passing and in pushing or popping a stack greatly improves the target code. The statements like $x = x + 1$ or $x = x - 1$ can also be replaced by these addressing modes. This technique is also called use of machine idioms.

The difficulty with this technique is that there are many ways to perform a computation on a typical target machine.

CD 6.11

**Mention the Issues to be considered while applying the techniques for code optimization.**

**Answer :** (Model Paper-II, Q1(f) | Nov.-15, Set-1, Q1(f))

Optimizing a program is required, so that the compiler can generate.

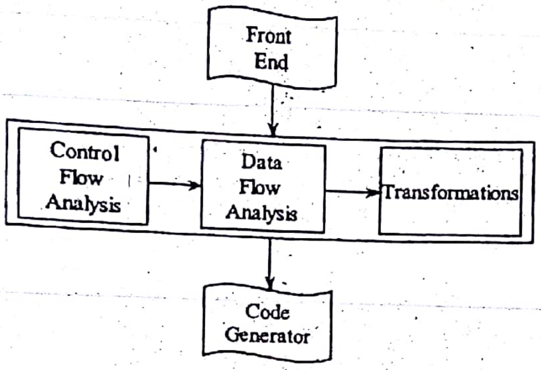
(i) Efficient target code along with preserving the meaning of the program

(ii) Reduction in the size of generated code.

Thus, getting better performance in terms of space and time, is the idea behind optimization.

❖ A compiler which spends some more amount of time on code optimization, when compared to the rest phases is known as an "optimizing compiler".

❖ The organization of an optimizing compiler is given in the figure below,



**Figure**

The code optimizer in the above figure uses the intermediate code and,

❖ Control flow analysis.

❖ Data flow analysis.

❖ Transformations to generate an optimized code.

❖ The input and output of the code optimizer is an intermediate code. However, the code is an optimized set of intermediate statements. In between, the code optimizer uses control flow analysis and then data flow analysis and then performs some transformations to generate optimized code.

❖ There can be situations where a piece of code cannot be optimized. Hence, the output from the code optimizer is the same set of intermediate code statements.

**Q6. Distinguish between machine dependent and machine independent optimization.**

**Answer :**

| | Machine Dependent | | Machine Independent |
|---|---|---|---|
| 1. | It is dependent on the instruction set and addressing modes to be used. | 1. | It is independent of the target machine, but depends on the source language characteristics. |
| 2. | The efficiency of the program is improved by allocating sufficient number of resources. | 2. | The efficiency of the target code is improved by using appropriate program structure. |
| 3. | Intermixed instructions with data increases the speed of execution. | 3. | Elimination of dead code increases the speed of execution. |
| 4. | Intermediate instructions are used wherever necessary. | 4. | Identical computations are moved to one place, thus avoiding the repeated computation of an expression. |

**Q7. Write a short note on,**

**(i) Points**

**(ii) Paths.**

**Answer :**

**(i) Points**

A point is the one that comes between two contiguous statements and the statements that comes one after the other. The following figure illustrates the use of points in basic block.

```
            pt1
1.  j : = j + 1
            pt2
2.  t2 : = 4 * j
            pt3
3.  t3 : = a[t2]
            pt4
4.  if t3 < pivot
            pt5
```

**Figure: Basic Block Containing Different Points**

As shown in above figure, if block contains four statements then there exists exactly five points (i.e., $pt_1, pt_2, pt_3, pt_4, pt_5$).

**(ii) Paths**

A path is considered as a set of points, say, $pt_1, pt_2, ...., pt_n$ which are arranged in sequential order providing the global view of the flow of control and satisfying one of the following two conditions for each '$i$' where $1 \leq i \leq n$.

1. If point '$pt_i$' comes exactly before the statement then point '$pt_{i+1}$' comes exactly after the statement in the same block, instead of two other blocks.
2. If point '$pt_i$' comes exactly at the end of one block then point '$pt_{i+1}$' comes exactly at the beginning of the next block or following block, instead of the same block.

**Q8. Explain different types of transformations used to improve the code.**

**Answer :**

**Model Paper-III, Q1(f)**

The different types of transformations used for improving the code are as follows,

1. Function preserving transformations
2. Structure preserving transformations
3. Algebraic transformations.

**1. Function Preserving Transformations**

These are the transformations carried out without making change in the computing function. In general, these are applied on global transformations.

**2. Structure Preserving Transformations**

These are the transformations carried out without making change in the set of expressions it computes. Usually, these are applied on local optimizations.

**3. Algebraic Transformations**

These are the transformations carried out to simplify the process of computation for set of expressions using algebraic identities. In this, expensive operations like multiplication by 2 is replaced with cheaper one i.e., left shift.