UNIT - IV

Run-Time Environments: Stack Allocation of Space, Access to Non-local Data on the Stack, Heap Management, Introduction to Garbage Collection, Introduction to Trace-Based Collection.

Code Generation: Issues in the Design of a Code Generator, The Target Language, Addresses in the Target Code, Basic Blocks and Flow Graphs, Optimization of Basic Blocks, A Simple Code Generator, Peephole Optimization, Register Allocation and Assignment, Dynamic Programming Code-Generation.

4.1 STACK ALLOCATION OF SPACE

Almost all compilers for languages that use procedure, functions, or methods manage their run-time memory as a stack. Whenever a procedure is called, the local variable's space is pushed into a stack and popped from the stack when the procedure terminates.

Actual parameters
Returned values
Control link(Dynamic Link)
Access link(Static Link)
Saved machine status
Local data
Temporaries

Activation Records

A run-time stack known as a control stack manages the procedure calls and returns. The control stack stores the activation record of each live activation. The top of the stack will hold the latest activation record.

The content of the activation record is given below. This record may vary according to the implemented languages.

Temporaries: The values which arise from the evaluation of expression will be held by temporaries.

Local data: The data belonging to the execution of the procedure is stored in local data.

Saved machine status: The status of the machine that might contain register, program counter before the call to the procedure is stored in saved machine status.

Access link: The data's information outside the local scope is stored in the access link.

Control link: The activation record of the caller is pointed by the control link.

Returned values: It represents the space for the return value of the called function if any.

Actual parameter: It represents the actual parameters used by the calling procedure.

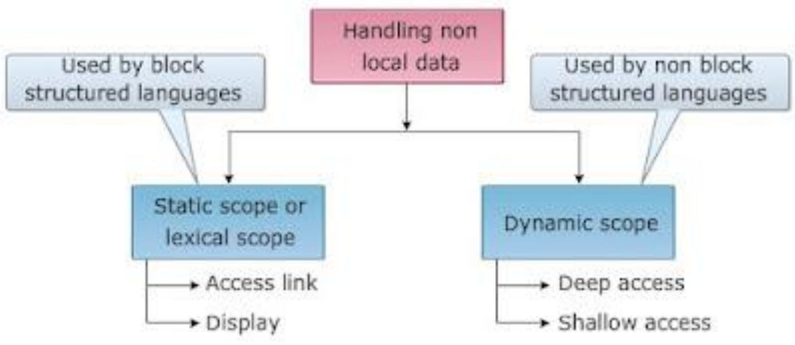
4.2 ACCESS TO NON-LOCAL NAMES:

In some cases, when a procedure refer to variables that are not local to it, then such variables are called non-local variables

There are two types of scope rules, for the non-local names. They are

Static scope

Dynamic scope



Handling non local data
Used by block structured languages
Used by non block structured languages
Static scope or lexical scope
Dynamic scope
Access link
Display
Deep access
Shallow access

Static Scope or Lexical Scope

Lexical scope is also called static scope. In this type of scope, the scope is verified by examining the text of the program. PASCAL, C and ADA are the languages that use the static scope rule.These languages are also called block structured languages.

Block

A block is a sequence of statements that contains the local data declarations. It is enclosed within the delimiters.

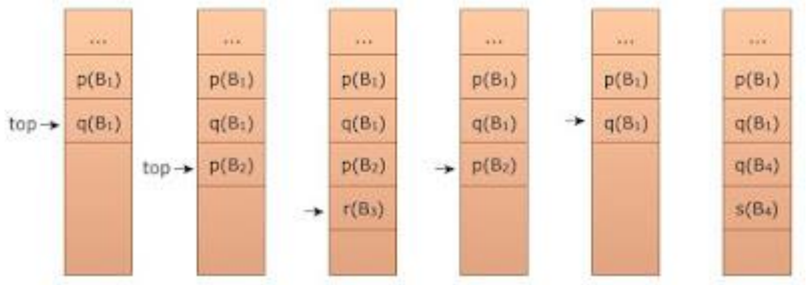
Example:

```
{
Declaration statements
……….
}
```

The beginning and end of the block are specified by the delimiter. The blocks can be in nesting fashion that means block B2 completely can be inside the block B1

The scope of declaration In a block structured language, is given by most closely nested loop or static rule.

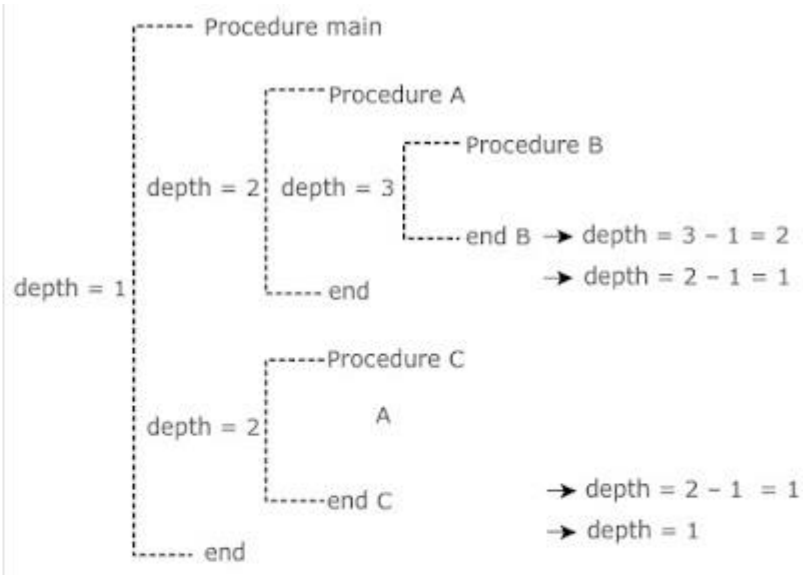
Example: obtain the static scope of the declarations made in the following piece of code.



Lexical Scope for Nested Procedure

If a procedure is declared inside another procedure then that procedure is known as nested procedure. A procedure pi, can call any procedure, i.e., its direct ancestor or older siblings of its direct ancestor

Procedure main
Procedure P1
Procedure P2
Procedure P3
Procedure P4



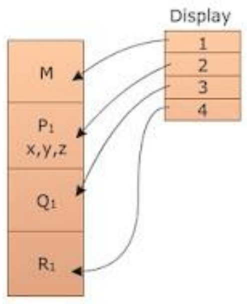
Access Link:

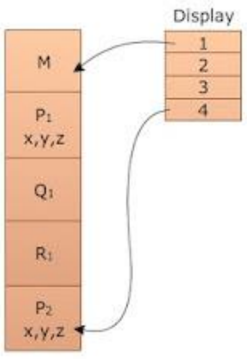
Access links are the pointers used in the implementation of lexical scope which is obtained by using pointer to each activation record. If procedure p is nested within a procedure q then access link of p points to access link or most recent activation record of procedure q

Displays:

If access links are used in the search, then the search can be slow So, optimization is used to access an activation record from the direct location of the variable without any search. Display is a global array d of pointers to activation records, indexed by lexical nesting depth. The number of display elements can be known at compiler time d[i] is an array element which points to the most recent activation of the block at nesting depth (or lexical level).

Example:





4.3 STORAGE ALLOCATION

Runtime environment manages runtime memory requirements for the following entities:

Code : It is known as the text part of a program that does not change at runtime. Its memory requirements are known at the compile time.

Procedures : Their text part is static but they are called in a random manner. That is why, stack storage is used to manage procedure calls and activations.

Variables : Variables are known at the runtime only, unless they are global or constant. Heap memory allocation scheme is used for managing allocation and de-allocation of memory for variables in runtime.

Static Allocation

In this allocation scheme, the compilation data is bound to a fixed location in the memory and it does not change when the program executes. As the memory requirement and storage locations are known in advance, runtime support package for memory allocation and de-allocation is not required.

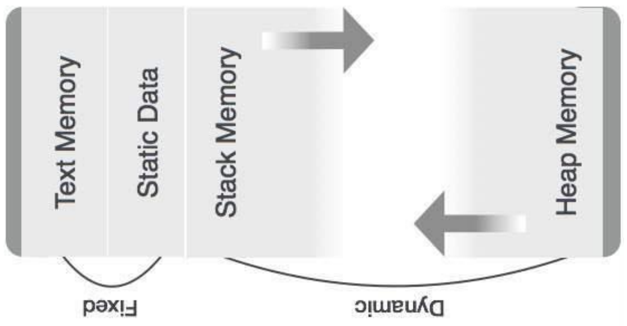
Stack Allocation

Procedure calls and their activations are managed by means of stack memory allocation. It works in last-in-first-out (LIFO) method and this allocation strategy is very useful for recursive procedure calls.

Heap Allocation

Variables local to a procedure are allocated and de-allocated only at runtime. Heap allocation is used to dynamically allocate memory to the variables and claim it back when the variables are no more required.

Except statically allocated memory area, both stack and heap memory can grow and shrink dynamically and unexpectedly. Therefore, they cannot be provided with a fixed.



As shown in the image above, the text part of the code is allocated a fixed amount of memory. Stack and heap memory are arranged at the extremes of total memory allocated to the program

4.4 INTRODUCTION TO GARBAGE COLLECTION

Garbage collection (GC) is a dynamic approach to automatic memory management and heap allocation that processes and identifies dead memory blocks and reallocates storage for reuse. The primary purpose of garbage collection is to reduce memory leaks.

Garbage collection (GC) is implementation requires three primary approaches, as follows:

- **Mark-and-sweep** - In process when memory runs out, the GC locates all accessible memory and then reclaims available memory.
- **Reference counting** - Allocated objects contain a reference count of the referencing number. When the memory count is zero, the object is garbage and is then destroyed. The freed memory returns to the memory heap.
- **Copy collection** - There are two memory partitions. If the first partition is full, the GC locates all accessible data structures and copies them to the second partition, compacting memory after GC process and allowing continuous free memory.

Garbage collection's dynamic approach to automatic heap allocation addresses common and costly errors that often result in real world program defects when undetected.

Garbage collection (GC) is not perfect, and the following drawbacks should be considered:

- When freeing memory, GC consumes computing resources.
- The GC process is unpredictable, resulting in scattered session delays.
- When unused object references are not manually disposed, GC causes logical memory leaks.

- GC does not always know when to process within virtual memory environments of modern desktop computers.
- The GC process interacts poorly with cache and virtual memory systems, resulting in performance-tuning difficulties.

4. 5 INTRODUCTION TO TRACE-BASED COLLECTION

Trace-based collectors will run at intervals to find unreachable objects(garbage) and reclaim their space. These intervals run whenever the free space goes below a set threshold or it is depleted.

Types of Trace-Based Garbage Collectors

Mark-and-Sweep: Marks each reachable node in first phase Releases every unmarked node in the second phase.

Copying Collector: Heap is divided into two spaces, only one of which is active Copies reachable objects from active to inactive space. Switches active and inactive spaces when copying is done

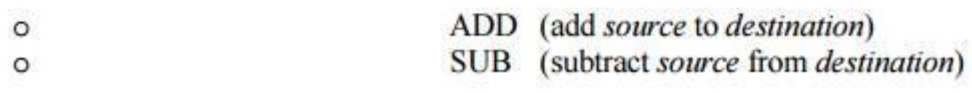
Generational GC: Objects are divided into old and new generations, and new generations are collected more often.

Concurrent GC: Garbage collector runs concurrently (e.g. in a separate thread) with the program; the program is not interrupted for collection.

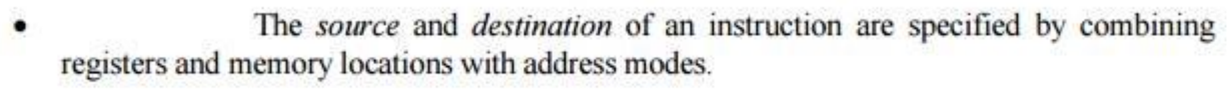
4.6 TARGET MACHINE

Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.

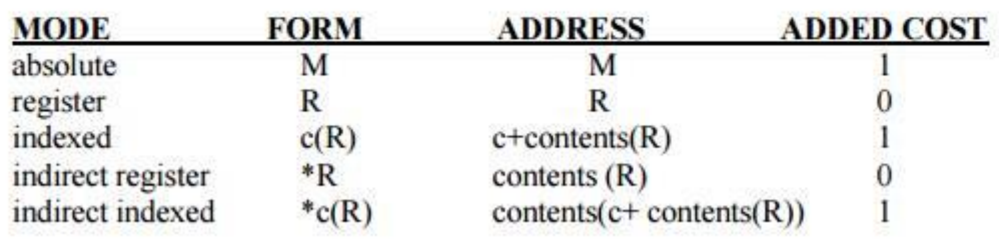
- The target computer is a byte-addressable machine with 4 bytes to a word.
- It has *n*general-purpose registers, R0, R1, . . . , Rn-1.
- It has two-address instructions of the form:



- ADD (add *source* to *destination*)
- SUB (subtract *source* from *destination*)



- The *source* and *destination* of an instruction are specified by combining registers and memory locations with address modes.



MODE	FORM	ADDRESS	ADDED COST
absolute	M	M	1
register	R	R	0
indexed	c(R)	c+contents(R)	1
indirect register	*R	contents (R)	0
indirect indexed	*c(R)	contents(c+ contents(R))	1

4.7 ISSUES IN THE DESIGN OF A CODE GENERATOR

Code generator converts the intermediate representation of source code into a form that can be readily executed by the machine. A code generator is expected to generate the correct code. Designing of code generator should be done in such a way so that it can be easily implemented, tested and maintained.

The following issue arises during the code generation phase:

Input to code generator –

The input to code generator is the intermediate code generated by the front end, along with information in the symbol table that determines the run-time addresses of the data-objects denoted by the names in the intermediate representation. Intermediate codes may be represented mostly in quadruples, triples, indirect triples, Postfix notation, syntax trees, DAG’s, etc. The code generation phase just proceeds on an assumption that the input are free from all of syntactic and state semantic errors, the necessary type checking has taken place and the type-conversion operators have been inserted wherever necessary.

Target program –

- The target program is the output of the code generator. The output may be absolute machine language, relocatable machine language, assembly language.
- Absolute machine language as output has advantages that it can be placed in a fixed memory location and can be immediately executed.
- Relocatable machine language as an output allows subprograms and subroutines to be compiled separately. Relocatable object modules can be linked together and loaded by linking loader. But there is added expense of linking and loading.
- Assembly language as output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code. And we need an additional assembly step after code generation.

Memory Management –

Mapping the names in the source program to the addresses of data objects is done by the front end and the code generator. A name in the three address statements refers to the symbol table entry for name. Then from the symbol table entry, a relative address can be determined for the name.

Instruction selection –

Selecting the best instructions will improve the efficiency of the program. It includes the instructions that should be complete and uniform. Instruction speeds and machine idioms also plays a major role when efficiency is considered. But if we do not care about the efficiency of the target program then instruction selection is straight-forward.

For example, the respective three-address statements would be translated into the latter code sequence as shown below:

```
P:=Q+R
S:=P+T

MOV Q, R0
ADD R, R0
MOV R0, P
MOV P, R0
ADD T, R0
MOV R0, S
```

Here the fourth statement is redundant as the value of the P is loaded again in that statement that just has been stored in the previous statement. It leads to an inefficient code sequence. A given intermediate representation can be translated into many code sequences, with significant cost differences between the different implementations. A prior knowledge of instruction cost is needed in order to design good sequences, but accurate cost information is difficult to predict.

Register allocation issues –

- Use of registers make the computations faster in comparison to that of memory, so efficient utilization of registers is important. The use of registers are subdivided into two subproblems:
- During Register allocation – we select only those set of variables that will reside in the registers at each point in the program.
- During a subsequent Register assignment phase, the specific register is picked to access the variable.

As the number of variables increases, the optimal assignment of registers to variables becomes difficult. Mathematically, this problem becomes NP-complete. Certain machine requires register pairs consist of an even and next odd-numbered register. For example

M a, b

These types of multiplicative instruction involve register pairs where the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

Evaluation order –

The code generator decides the order in which the instruction will be executed. The order of computations affects the efficiency of the target code. Among many computational orders, some will require only fewer registers to hold the intermediate results. However, picking the best order in the general case is a difficult NP-complete problem.

Approaches to code generation issues: Code generator must always generate the correct code. It is essential because of the number of special cases that a code generator might face. Some of the design goals of code generator are:

- Correct
- Easily maintainable
- Testable
- Efficient

4.8 BASIC BLOCKS AND FLOW GRAPHS

The basic block is a set of statements. The basic blocks do not have any in and out branches except entry and exit. It means the flow of control enters at the beginning and will leave at the end without any halt. The set of instructions of basic block executes in sequence.

Here, the first task is to partition a set of three-address code into the basic block. The new basic block always starts from the first instruction and keep adding instructions until a jump or a label is met. If no jumps or labels are found, the control will flow in sequence from one instruction to another.

The algorithm for the construction of basic blocks is given below:

Algorithm: Partitioning three-address code into basic blocks.
Input: The input for the basic blocks will be a sequence of three-address code.
Output: The output is a list of basic blocks with each three address statements in exactly one block.

Method: First identify the leader in the code. The rules for finding leaders are as follows:

- The first statement is a leader.
- Statement L is a leader if there is an conditional or unconditional goto statement like: if....goto L or goto L
- Instruction L is a leader if it immediately follows a goto or conditional goto statement like: if goto B or goto B

Each leader's basic block will have all the instructions from the leader itself until the instruction, which is just before the starting of the next leader.

Consider the following code construct basic blocks

```
begin
        prod :=0;
i:=1;
do
prod :=prod+ a[i] * b[i];
i :=i+1;
```

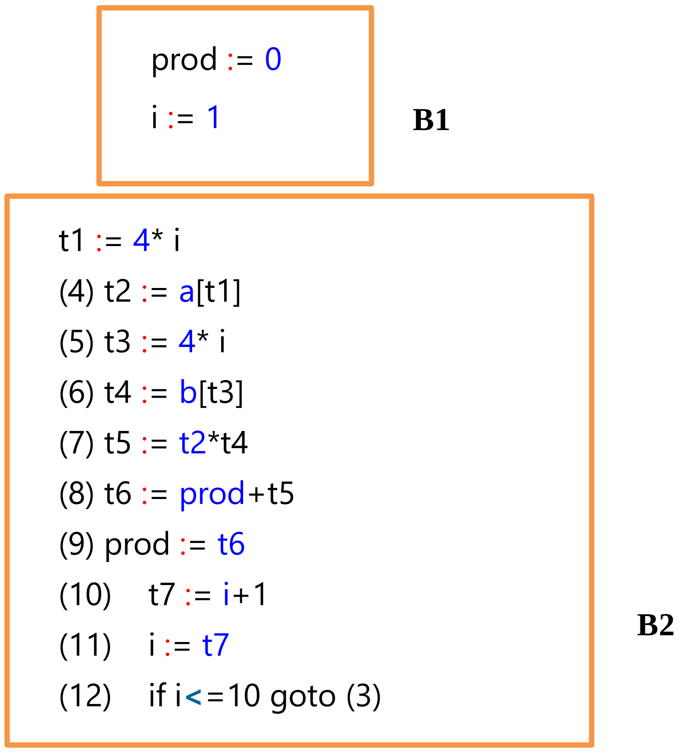
```
end
while i <= 10
end
```

Sol:

Step1: Three Address Code

```
(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<=10 goto (3)
```

Step2 : Basic Blocks



```
prod := 0
i := 1
```

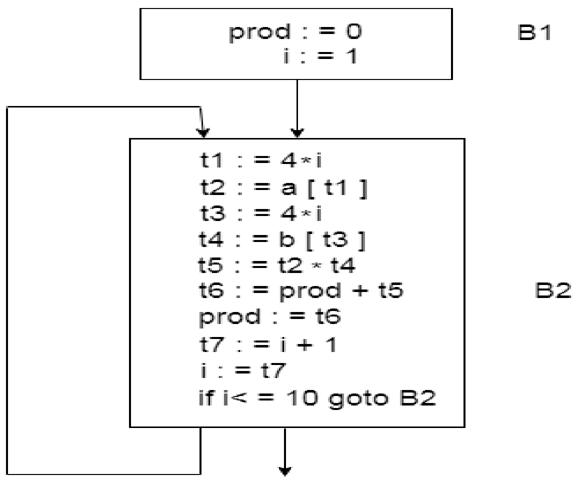
B1

```
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<=10 goto (3)
```

B2

Flow Graph
Flow graph is a directed graph. It contains the flow of control information for the set of

basic block. A control flow graph is used to depict that how the program control is being parsed among the blocks. It is useful in the loop optimization.
Flow graph for the vector dot product is given as follows:



- Block B1 is the initial node. Block B2 immediately follows B1, so from B2 to B1 there is an edge.
- The target of jump from last statement of B1 is the first statement B2, so from B1 to B2 there is an edge.
- B2 is a successor of B1 and B1 is the predecessor of B2.

4.9 OPTIMIZATION OF BASIC BLOCKS

Optimization process can be applied on a basic block. While optimization, we don't need to change the set of expressions computed by the block.

There are two type of basic block optimization. These are as follows:

1. Structure-Preserving Transformations
2. Algebraic Transformations

1. Structure-Preserving Transformations
The primary Structure-Preserving Transformation on basic blocks are as follows:

- Common sub-expression elimination
- Dead code elimination

Common sub-expression elimination:

Suppose we have two expressions that compute the same values. In that case, we need to eliminate one of the expressions. This method is known as the common sub-expression elimination method.

Example:

```
p = a + b
q = c – d
r = x+ y
s = c - d
```

This example shows that the second and fourth statements compute the same expression: c - d. So the basic block can be transformed as follows:

```
p = a + b
q = c - d
r = x + y
s = q
```

Dead code elimination:

Whenever a programmer writes a program, there is a possibility that the program may have some dead code. These dead codes are the result of some expression in the program. Generally, a programmer doesn't introduce dead code intentionally. The dead code may be a variable or the result of some expression computed by the programmer that may not have any further uses. By eliminating these useless things from a code, the code will get optimized.

Example:

A statement p = q + r appears in a block, and p is a dead symbol. It means that it will never be used subsequently so that we can eliminate this statement. This elimination does not have any impact on the values of the basic block.

2. Algebraic Transformations:

We can also optimized the basic blocks using algebraic identities. For example, we may apply arithmetic identities, such as

Local reduction in strength is also another kind of algebraic transformation. In this optimization, a more expensive operator is replaced by a cheaper one.

x^2	Instead of	x * x
2 * x	Instead of	x + x
x/2	Instead of	x * 0.5

4.10 A SIMPLE CODE GENERATOR

- A code generator generates target code for a sequence of three- address statements and
  effectively uses registers to store operands of the statements.
- For example: consider the three-address statement a := b+c It can have the following sequence of codes:

```
ADD Rj, Ri Cost = 1
        (or)
ADD c, Ri Cost = 2
        (or)
MOV c, Rj Cost = 3
ADD Rj, Ri
```

Register and Address Descriptors:

- A register descriptor is used to keep track of what is currently in each registers. The register
  descriptors show that initially all the registers are empty.
- An address descriptor stores the location where the current value of the name can be found at run time.

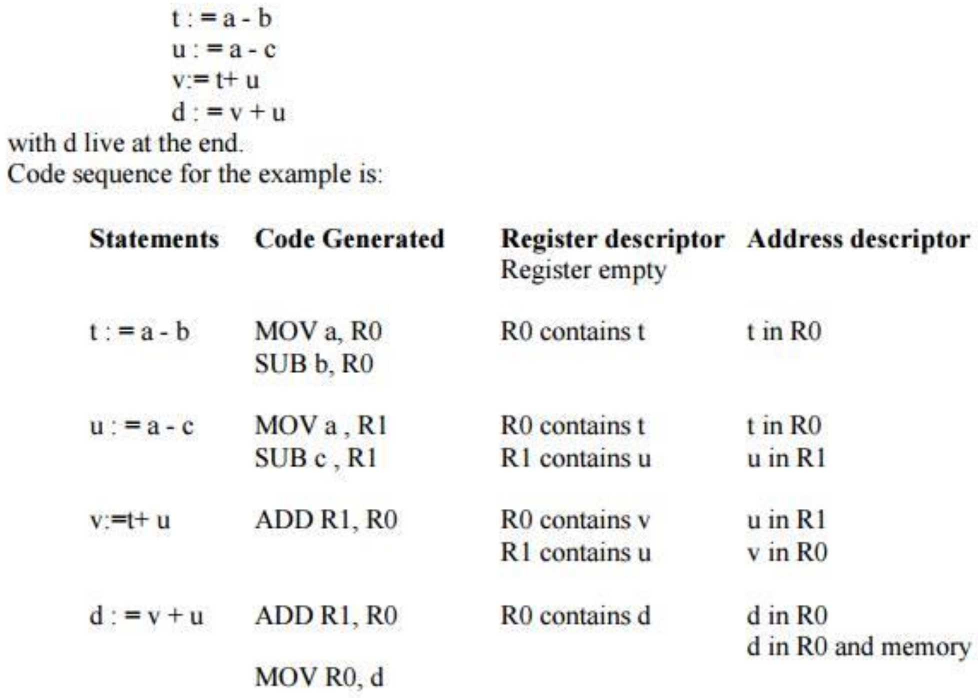
A code-generation algorithm:

The algorithm takes as input a sequence of three-address statements constituting a basic block. For each three-address statement of the form x : = y op z, perform the following actions:

1. Invoke a function get reg to determine the location L where the result of the computation y op z should be stored.
2. Consult the address descriptor for y to determine y', the current location of y. Prefer the register for y' if the value of y is currently both in memory and a register. If the value of y is not already in L, generate the instruction MOV y' , L to place a copy of y in L
3. Generate the instruction OP z' , L where z' is a current location of z. Prefer a register to a memory location if z is in both. Update the address descriptor of x to indicate that x is in location L. If x is in L, update its descriptor and remove x from all other descriptors.
4. If the current values of y or z have no next uses, are not live on exit from the block, and are in registers, alter the register descriptor to indicate that, after execution of x : = y op z , those registers will no longer contain y or z

Generating Code for Assignment Statements:

• The assignment d : = (a-b) + (a-c) + (a-c) might be translated into the following three-address code sequence:
Code sequence for the example is:



```
t : = a - b
u : = a - c
v:= t+ u
d : = v + u
```

with d live at the end.
Code sequence for the example is:

Statements	Code Generated	Register descriptor	Address descriptor
		Register empty	
t : = a - b	MOV a, R0 SUB b, R0	R0 contains t	t in R0
u : = a - c	MOV a , R1 SUB c , R1	R0 contains t R1 contains u	t in R0 u in R1
v:=t+ u	ADD R1, R0	R0 contains v R1 contains u	u in R1 v in R0
d : = v + u	ADD R1, R0 MOV R0, d	R0 contains d	d in R0 d in R0 and memory

4.11 REGISTER ALLOCATION AND ASSIGNMENT

Local register allocation

Register allocation is only within a basic block. It follows top-down approach.
Assign registers to the most heavily used variables

- Traverse the block
- Count uses
- Use count as a priority function
- Assign registers to higher priority variables first

Advantage

Heavily used values reside in registers

Disadvantage

Does not consider non-uniform distribution of uses

Need of global register allocation

Local allocation does not take into account that some instructions (e.g. those in loops) execute more frequently. It forces us to store/load at basic block endpoints since each block has no knowledge of the context of others.

To find out the live range(s) of each variable and the area(s) where the variable is used/defined global allocation is needed. Cost of spilling will depend on frequencies and locations of uses.

Register allocation depends on:
- Size of live range
- Number of uses/definitions
- Frequency of execution
- Number of loads/stores needed.
- Cost of loads/stores needed.

Register allocation by graph coloring
Global register allocation can be seen as a graph coloring problem.
Basic idea:
1. Identify the live range of each variable
2. Build an interference graph that represents conflicts between live ranges (two nodes are connected if the variables they represent are live at the same moment)
3. Try to assign as many colors to the nodes of the graph as there are registers so that two neighbors have different colors

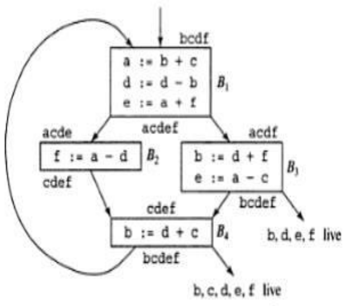


Fig 4.3 Flow graph of an inner loop

4.12 PEEPHOLE OPTIMIZATION
A statement-by-statement code-generations strategy often produces target code that contains redundant instructions and suboptimal constructs. The quality of such target code can be improved by applying “optimizing” transformations to the target program.

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, whenever possible.

The peephole is a small, moving window on the target program. The code in the peephole need not be contiguous, although some implementations do require this. It is characteristic of peephole optimization that each improvement may spawn opportunities for additional improvements.

Characteristics of peephole optimizations:

- Redundant-instructions elimination
- Flow-of-control optimizations
- Algebraic simplifications
- Use of machine idioms
- Unreachable

Redundant Loads And Stores:

If we see the instructions sequence

(1) MOV R0,a

(2) MOV a,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).

Unreachable Code:

Another opportunity for peephole optimizations is the removal of unreachable instructions. An unlabeled instruction immediately following an unconditional jump may be removed. This operation can be repeated to eliminate a sequence of instructions. For example, for debugging purposes, a large program may have within it certain segments that are executed only if a variable debug is 1. In C, the source code might look like:

```
#define debug 0
….

If ( debug ) {
Print debugging information

}
```

In the intermediate representations the if-statement may be translated as:

```
If debug =1 goto L1 goto L2

L1: print debugging information L2: ………………………… (a)
```

One obvious peephole optimization is to eliminate jumps over jumps .Thus no matter what the value of debug; (a) can be replaced by:

```
        If debug ≠1 goto L2
        Print debugging information
L2: ……………………………… (b)

        If debug ≠0 goto L2
        Print debugging information
L2: ……………………………… (c)
```

As the argument of the statement of (c) evaluates to a constant true it can be replaced

By goto L2. Then all the statement that print debugging aids are manifestly unreachable and can be eliminated one at a time.

Flows-Of-Control Optimizations:

The unnecessary jumps can be eliminated in either the intermediate code or the target code by the following types of peephole optimizations. We can replace the jump sequence

```
        goto L1
        ….

L1: gotoL2 (d)
```

by the sequence

```
        goto L2
        ….

L1: goto L2
```

If there are now no jumps to L1, then it may be possible to eliminate the statement L1:goto L2 provided it is preceded by an unconditional jump .Similarly, the sequence

```
if a < b goto L1
….

L1: goto L2 (e)
```

can be replaced by

If a < b goto L2

….

L1: goto L2

Ø Finally, suppose there is only one jump to L1 and L1 is preceded by an unconditional goto. Then the sequence

goto L1

L1: if a < b goto L2 (f) L3:

may be replaced by

If a < b goto L2
goto L3

…….

L3:

While the number of instructions in(e) and (f) is the same, we sometimes skip the unconditional jump in (f), but never in (e).Thus (f) is superior to (e) in execution time

Algebraic Simplification:

There is no end to the amount of algebraic simplification that can be attempted through peephole optimization. Only a few algebraic identities occur frequently enough that it is worth considering implementing them. For example, statements such as

x := x+0 or
x := x * 1

are often produced by straightforward intermediate code-generation algorithms, and they can be eliminated easily through peephole optimization.

Reduction in Strength:

Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.

For example, x^2 is invariably cheaper to implement as x*x than as a call to an exponentiation routine. Fixed-point multiplication or division by a power of two is cheaper to implement as a shift. Floating-point division by a constant can be implemented as multiplication by a constant, which may be cheaper.

X2 → X*X

Use of Machine Idioms:

The target machine may have hardware instructions to implement certain specific operations efficiently. For example, some machines have auto-increment and auto-decrement addressing modes. These add or subtract one from an operand before or after using its value. The use of these modes greatly improves the quality of code when pushing or popping a stack, as in parameter passing. These modes can also be used in code for statements like i : =i+1.

i:=i+1 → i++
i:=i-1 → i- -

4.13 DAG FOR REGISTER ALLOCATION

Code generation from DAG is much simpler than the linear sequence of three address code

With the help of DAG one can rearrange sequence of instructions and generate and efficient code

There exist various algorithms which are used for generating code from DAG. They are:

Code Generation from DAG:-

Rearranging Order
Heuristic Ordering
Labeling Algorithm

Rearranging Order

These address code's order affects the cost of the object code which is being generated

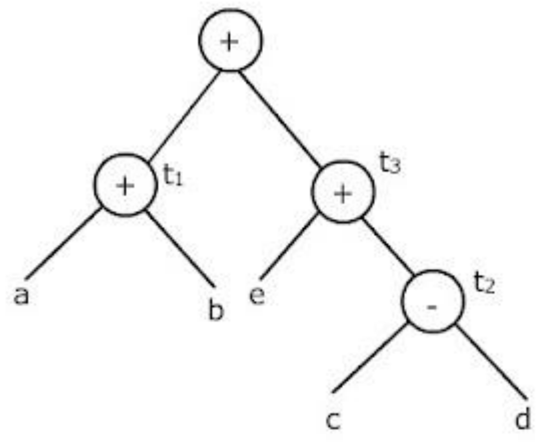
Object code with minimum cost can be achieved by changing the order of computations

Example:

t1:= a + b
t2:= c – d
t3:= e + t2

t4:= t1 + t3

For the expression (a+b) + (e+(c-d)), a DAG can be constructed for the above sequence as shown below



DAG for *(a + b) + (e + (c - d))*

The code is thus generated by translating the three address code line by line

```
MOV a, R0
ADD b, R0
MOV c, R1
SUB d, R1
MOV R0, t          t1:= a+b
MOV e, R0          R1 has c-d
ADD R0, R1           /* R1 contains e + (c – d)*/
MOV t1, R0         /R0 contains a + b*/
ADD R1, R0
MOV R0, t4
```

Now, if the ordering sequence of the three address code is changed

```
t2:= c - d
t3:= e + t2
t1:= a + b
t4:= t1 + t3
```

Then, an improved code is obtained as:

```
MOV c, R0
SUB D, R0
MOV e, R1
ADD R0, R1
MOV a, R0
ADD b, R0
ADD R1, R0
MOV R0, t4
```

Heuristic Ordering

The algorithm displayed below is for heuristic ordering. It lists the nodes of a DAG such that the node’s reverse listing results in the computation order.

```
{
While unlisted interior nodes remain
do
{
select an unlisted node n, all of whose parents have been listed;
List n;
While the leftmost child ‘m’ of ‘n’ has no unlisted parents and is not a leaf
do
{
/*since na was just listed, surely m is not yet listed*/
}
{
list m
n =m;
}
}
order = reverse of the order of listing of nodes
}
```

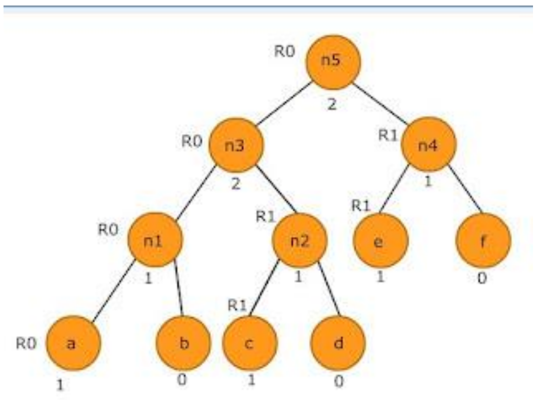
Labeling Algorithm

Labeling algorithm deals with the tree representation of a sequence of three address statements

It could similarly be made to work if the intermediate code structure was a parse tree. This algorithm has two parts:

- The first part labels each node of the tree from the bottom up, with an integer that denotes the minimum number of registers required to evaluate the tree, and with no storing of intermediate results
- The second part of the algorithm is a tree traversal that travels the tree in an order governed by the computed labels in the first part, and which generates the code during the tree traversal

```
if n is a leaf then
        if n is leftmost child of its parents then
                Label(n) = 1
        else label(n) = 0
else
{
        /*n is an interior node*/
        let n1, n2, …..,nk be the children of ordered by label,
        so label(n1) >= label(n2) >= … >= label(nk);
        label(n) = max (label (ni) + i – 1)
}
```



4.15 THE DAG REPRESENTATION FOR BASIC BLOCKS

A DAG for a basic block is a directed acyclic graph with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or constants.
2. Interior nodes are labeled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels to store the computed values.

- DAGs are useful data structures for implementing transformations on basic blocks.
- It gives a picture of how the value computed by a statement is used in subsequent statements.
- It provides a good way of determining common sub - expressions.

Algorithm for construction of DAG

Input: A basic block

Output: A DAG for the basic block containing the following information:

1. A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.
2. For each node a list of attached identifiers to hold the computed values. Case (i) x : = y OP z Case (ii) x : = OP y
3. Case (iii) x : = y

Method:

Step 1:

If y is undefined then create node(y).

If z is undefined, create node(z) for case(i).

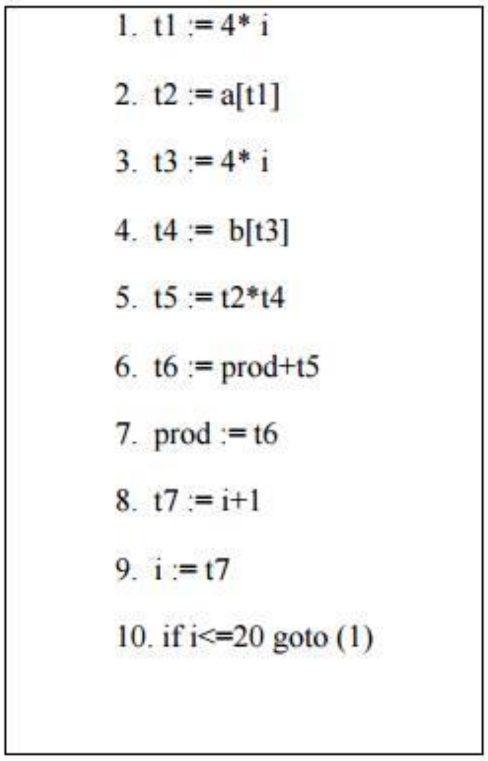
Step 2:

- For the case(i), create a node(OP) whose left child is node(y) and right child is node(z). (Checking for common sub expression). Let n be this node.

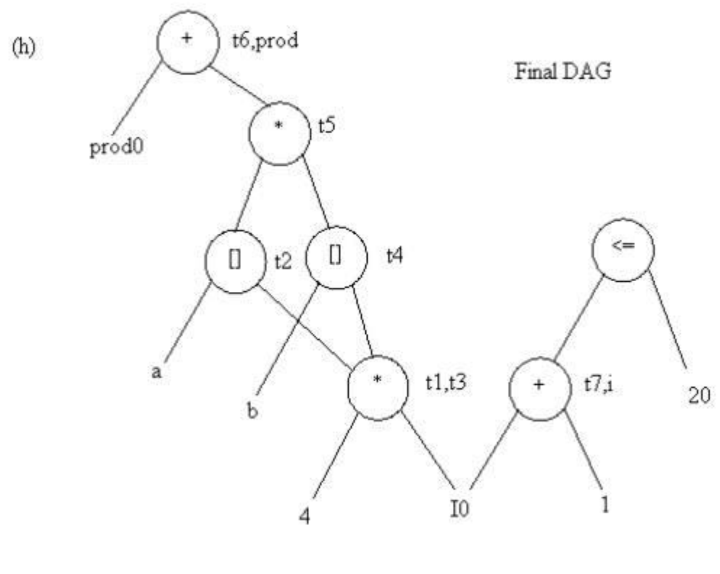
- For case(ii), determine whether there is node(OP) with one child node(y). If not create such a node.
- For case(iii), node n will be node(y).

Step 3:
Delete x from the list of identifiers for node(x). Append x to the list of attached identifiers for the node n found in step 2 and set node(x) to n.

Example: Consider the block of three- address statements in Fig



```
1. t1 := 4* i
2. t2 := a[t1]
3. t3 := 4* i
4. t4 :=  b[t3]
5. t5 := t2*t4
6. t6 := prod+t5
7. prod := t6
8. t7 := i+1
9. i := t7
10. if i<=20 goto (1)
```



Application of DAGs:

1. We can automatically detect common sub expressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values that could be used outside the block.