PUNE VIDYARTHI GRIHA's

COLLEGE OF ENGINEERING

(Approved by AICTE, Accredited by NAAC, Affiliated to SPPU)

NASHIK – 04.

COMPILER NOTES

UNIT - V

DEPARTMENT OF COMPUTER ENGINEERING

AY – 2018-19

UNIT – V

Syllabus - Code Generation - Issues in code generation, basic blocks, flow graphs, DAG representation of basic blocks, Target machine description, peephole optimization, Register allocation and Assignment, Simple code generator, Code generation from labeled tree, Concept of code generator.

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 a 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:

1. **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. Assume that they 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.

2. **Target Program :**
   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 an 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.
   - Assembly language as an output makes the code generation easier. We can generate symbolic instructions and use macro-facilities of assembler in generating code.
3. **Memory management :**
   Mapping the names in the source program to addresses of data objects is done by the front end and the code generator. A name in the three address statement refers to the symbol table entry for name.

Then from the symbol table entry, a relative address can be determined for the name.

4. **Instruction Selection :**

   Selecting 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 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.

5. **Register Allocation issue :**

   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:

   1. During **Register allocation –** we select only those set of variables that will reside in the registers at each point in the program.

2. During a subsequent **Register assignment** phase, the specific register is picked to access the variable.

As the number of variables increase, 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 a, the multiplicand is an even register and b, the multiplier is the odd register of the even/odd register pair.

6. **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 general case is a difficult NP-complete problem.

7. **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
   - Maintainable

7.2 BASIC BLOCKS AND FLOW GRAPHS

A graph representation of three-address statements, called a flow graph, is useful for understanding code-generation algorithms, even if the graph is not explicitly constructed by a code-generation algorithm. Nodes in the flow graph represent computations, and the edges represent the flow of control. Flow graph of a program can be used as a vehicle to collect information about the intermediate program. Some register-assignment algorithms use flow graphs to find the inner loops where a program is expected to spend most of its time.

BASIC BLOCKS

A basic block is a sequence of consecutive 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. The following sequence of three-address statements forms a basic block:

```
t1 := a*a
t2 := a*b
t3 := 2*t2
t4 := t1+t3
t5 := b*b
t6 := t4+t5
```

A three-address statement x := y+z is said to define x and to use y or z. A name in a basic block is said to live at a given point if its value is used after that point in the program, perhaps in another basic block.

The following algorithm can be used to partition a sequence of three-address statements into basic blocks.

Algorithm 1: Partition 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:

1. We first determine the set of leaders, the first statements of basic blocks.

The rules we use are the following:

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.

2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

Example 3: Consider the fragment of source code shown in fig. 7; it computes the dot product of two vectors a and b of length 20. A list of three-address statements performing this computation on our target machine is shown in fig. 8.

```
begin
prod := 0;
i := 1;
do begin
prod := prod + a[i] * b[i];
i := i+1;
end
while i<= 20
end
```

Let us apply Algorithm 1 to the three-address code in fig 8 to determine its basic

blocks. statement (1) is a leader by rule (I) and statement (3) is a leader by rule (II), since the last statement can jump to it. By rule (III) the statement following (12) is a leader. Therefore, statements (1) and (2) form a basic block. The remainder of the program beginning with statement (3) forms a second basic block.

```
(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)
```

7.3 TRANSFORMATIONS ON BASIC BLOCKS

A basic block computes a set of expressions. These expressions are the values of the names live on exit from block. Two basic blocks are said to be equivalent if they compute the same set of expressions. A number of transformations can be applied to a basic block without changing the set of expressions computed by the block. Many of these transformations are useful for improving the quality of code that will be ultimately generated from a basic block. There are two important classes of local transformations that can be applied to basic blocks; these are the structure-preserving transformations and the algebraic transformations.

7.4 STRUCTURE-PRESERVING TRANSFORMATIONS

The primary structure-preserving transformations on basic blocks are:

1. Common sub-expression elimination
2. Dead-code elimination
3. Renaming of temporary variables
4. Interchange of two independent adjacent statements

We assume basic blocks have no arrays, pointers, or procedure calls.

1. Common sub-expression elimination

Consider the basic block

a:= b+c

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

The second and fourth statements compute the same expression, namely b+c-d, and hence this basic block may be transformed into the equivalent block

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

Although the 1st and 3rd statements in both cases appear to have the same expression

on the right, the second statement redefines b. Therefore, the value of b in the 3rd statement is different from the value of b in the 1st, and the 1st and 3rd statements do not compute the same expression.

2. Dead-code elimination

Suppose x is dead, that is, never subsequently used, at the point where the statement x:= y+z appears in a basic block. Then this statement may be safely removed without changing the value of the basic block.

3. Renaming temporary variables

Suppose we have a statement t:= b+c, where t is a temporary. If we change this statement to u:= b+c, where u is a new temporary variable, and change all uses of this instance of t to u, then the value of the basic block is not changed.

4. Interchange of statements

Suppose we have a block with the two adjacent statements

t1:= b+c
t2:= x+y

Then we can interchange the two statements without affecting the value of the block if and only if neither x nor y is t1 and neither b nor c is t2. A normal-form basic block permits all statement interchanges that are possible.

7.5 DAG REPRESENTATION OF BASIC BLOCKS

The goal is to obtain a visual picture of how information flows through the block. The leaves will show the values entering the block and as we proceed up the DAG we encounter uses of these values defs (and redefs) of values and uses of the new values.
Formally, this is defined as follows.

1. Create a leaf for the initial value of each variable appearing in the block. (We do not know what that the value is, not even if the variable has ever been given a value).
2. Create a node N for each statement s in the block.

i. Label N with the operator of s. This label is drawn inside the node.

ii. Attach to N those variables for which N is the last def in the block. These additional labels

are drawn along side of N.
iii. Draw edges from N to each statement that is the last def of an operand used by N.

2. Designate as output nodes those N whose values are live on exit, an officially-mysterious term meaning values possibly used in another block. (Determining the live on exit values requires global, i.e., inter-block, flow analysis.) As we shall see in the next few sections various basic-block optimizations are facilitated by using the DAG.
Finding Local Common Subexpressions

As we create nodes for each statement, proceeding in the static order of the tatements, we might notice that a new node is just like one already in the DAG in which case we don't need a new node and can use the old node to compute the new value in addition to the one it already was computing. Specifically, we do not construct a new node if an existing node has

the same children in the same order and is labeled with the same operation.
Consider computing the DAG for the following block of code.

```
a = b + c
c = a + x
d = b + c
b = a + x
```

The DAG construction is explain as follows (the movie on the right accompanies the explanation).
1. First we construct leaves with the initial values.

2. Next we process a = b + c. This produces a node labeled + with a attached and having b0 and c0 as children.
3. Next we process c = a + x.

4. Next we process d = b + c. Although we have already computed b + c in the first statement, the c's are not the same, so we produce a new node.
5. Then we process b = a + x. Since we have already computed a + x in statement 2, we do not produce a new node, but instead attach b to the old node.
6. Finally, we tidy up and erase the unused initial values.

You might think that with only three computation nodes in the DAG, the block could be reduced to three statements (dropping the computation of b). However, this is wrong. Only if b is dead on exit can we omit the computation of b. We can, however, replace the last statement with the simpler b = c. Sometimes a combination of techniques finds

improvements that no single technique would find. For example if a-b is computed, then both a and b are incremented by one, and then a-b is computed again, it will not be recognized as a common subexpression even though the value has not changed. However, when combined with various algebraic transformations, the common value can be recognized.

Directed Acyclic Graph

Directed Acyclic Graph (DAG) is a tool that depicts the structure of basic blocks, helps to see the flow of values flowing among the basic blocks, and offers optimization too. DAG provides easy transformation on basic blocks. DAG can be understood here:

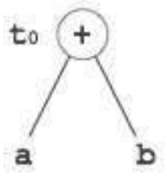
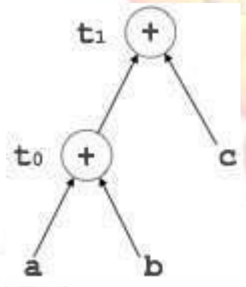
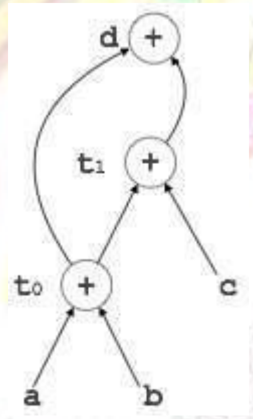
- Leaf nodes represent identifiers, names or constants.
- Interior nodes represent operators.
- Interior nodes also represent the results of expressions or the identifiers/name where the values are to be stored or assigned.

Example:

$t_0 = a + b$

$t_1 = t_0 + c$

$d = t_0 + t_1$

 [$t_0 = a + b$]	 [$t_1 = t_0 + c$]	 [$d = t_0 + t_1$]

Peephole Optimization

This optimization technique works locally on the source code to transform it into an optimized code. By locally, we mean a small portion of the code block at hand. These methods can be applied on intermediate codes as well as on target codes. A bunch of statements is analyzed and are checked for the following possible optimization:

Redundant instruction elimination

At source code level, the following can be done by the user:

int add_ten(int x)
{
int y, z;
y = 10;
z = x + y;
return z;
}
	int add_ten(int x)
	{
	int y;
	y = 10;
	y = x + y;
	return y;
	}
		int add_ten(int x)
		{
		int y = 10;
		return x + y;
		}
			int add_ten(int x)
			{
			return x + 10;
			}

At compilation level, the compiler searches for instructions redundant in nature. Multiple loading and storing of instructions may carry the same meaning even if some of them are removed. For example:

- MOV x, R0
- MOV R0, R1

We can delete the first instruction and re-write the sentence as:

```
MOV x, R1
```

Unreachable code

Unreachable code is a part of the program code that is never accessed because of programming constructs. Programmers may have accidently written a piece of code that can never be reached.

Example:

```
void add_ten(int x)
{
  return x + 10;
  printf("value of x is %d", x);
}
```

In this code segment, the **printf** statement will never be executed as the program control returns back before it can execute, hence **printf** can be removed.

Flow of control optimization

There are instances in a code where the program control jumps back and forth without performing any significant task. These jumps can be removed. Consider the following chunk of code:

```
...

MOV R1, R2

GOTO L1

...

L1 :   GOTO L2

L2 :   INC R1
```

In this code,label L1 can be removed as it passes the control to L2. So instead of jumping to L1 and then to L2, the control can directly reach L2, as shown below:

```
...



MOV R1, R2

GOTO L2

...

L2 :   INC R1
```

Algebraic expression simplification

There are occasions where algebraic expressions can be made simple. For example, the expression **a = a + 0** can be replaced by **a** itself and the expression a = a + 1 can simply be replaced by INC a.

Strength reduction

There are operations that consume more time and space. Their 'strength' can be reduced by replacing them with other operations that consume less time and space, but produce the same result.

For example, **x * 2** can be replaced by **x << 1**, which involves only one left shift. Though the output of a * a and a^2 is same, a^2 is much more efficient to implement.

Accessing machine instructions

The target machine can deploy more sophisticated instructions, which can have the capability to perform specific operations much efficiently. If the target code can accommodate those instructions directly, that will not only improve the quality of code, but also yield more efficient results.

Code Generator

A code generator is expected to have an understanding of the target machine's runtime environment and its instruction set. The code generator should take the following things into consideration to generate the code:

- **Target language** : The code generator has to be aware of the nature of the target language for which the code is to be transformed. That language may facilitate some machine-specific instructions to help the compiler generate the code in a more convenient way. The target machine can have either CISC or RISC processor architecture.
- **IR Type** : Intermediate representation has various forms. It can be in Abstract Syntax Tree (AST) structure, Reverse Polish Notation, or 3-address code.
- **Selection of instruction** : The code generator takes Intermediate Representation as input and converts (maps) it into target machine's instruction set. One representation can have many ways (instructions) to convert it, so it becomes the responsibility of the code generator to choose the appropriate instructions wisely.
- **Register allocation** : A program has a number of values to be maintained during the execution. The target machine's architecture may not allow all of the values to be kept in the CPU memory or registers. Code generator decides what values to keep in the registers. Also, it decides the registers to be used to keep these values.
- **Ordering of instructions** : At last, the code generator decides the order in which the instruction will be executed. It creates schedules for instructions to execute them.

Descriptors

The code generator has to track both the registers (for availability) and addresses (location of values) while generating the code. For both of them, the following two descriptors are used:

- **Register descriptor** : Register descriptor is used to inform the code generator about the availability of registers. Register descriptor keeps track of values stored in each register. Whenever a new register is required during code generation, this descriptor is consulted for register availability.
- **Address descriptor** : Values of the names (identifiers) used in the program might be stored at different locations while in execution. Address descriptors are used to keep track of memory locations where the values of identifiers are stored. These locations may include CPU registers, heaps, stacks, memory or a combination of the mentioned locations.

Code generator keeps both the descriptor updated in real-time. For a load statement, LD R1, x, the code generator:

- updates the Register Descriptor R1 that has value of x and
- updates the Address Descriptor (x) to show that one instance of x is in R1.

Code Generation

Basic blocks comprise of a sequence of three-address instructions. Code generator takes these sequence of instructions as input.

Note : If the value of a name is found at more than one place (register, cache, or memory), the register's value will be preferred over the cache and main memory. Likewise cache's value will be preferred over the main memory. Main memory is barely given any preference.

getReg : Code generator uses *getReg* function to determine the status of available registers and the location of name values. *getReg* works as follows:

- If variable Y is already in register R, it uses that register.
- Else if some register R is available, it uses that register.
- Else if both the above options are not possible, it chooses a register that requires minimal number of load and store instructions.

For an instruction x = y OP z, the code generator may perform the following actions. Let us assume that L is the location (preferably register) where the output of y OP z is to be saved:

- Call function getReg, to decide the location of L.
- Determine the present location (register or memory) of **y** by consulting the Address Descriptor of **y**. If **y** is not presently in register **L**, then generate the following instruction to copy the value of **y** to **L**:

  MOV y', L

  where **y'** represents the copied value of **y**.
- Determine the present location of **z** using the same method used in step 2 for **y** and generate the following instruction:

  OP z', L

  where **z'** represents the copied value of **z**.

- Now L contains the value of y OP z, that is intended to be assigned to **x**. So, if L is a register, update its descriptor to indicate that it contains the value of **x**. Update the descriptor of **x** to indicate that it is stored at location **L**.
- If y and z has no further use, they can be given back to the system.

Other code constructs like loops and conditional statements are transformed into assembly language in general assembly way.

7.6 PEEPHOLE OPTIMIZATION

A statement-by-statement code-generations strategy often produce target code that contains redundant instructions and suboptimal constructs .The quality of such target code can be improved by applying "optimizing" transformations to the 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 contiguous, although some implementations do require this. We shall give the following examples of program transformations that are characteristic of peephole optimizations:
• Redundant-instructions elimination

• Flow-of-control optimizations

• Algebraic simplifications

• Use of machine idioms

7.7 DEAD CODE ELIMINATION

Assume we are told (by global flow analysis) that certain values are dead on exit. We examine each root (node with no ancestor) and delete any that have no live variables attached. This process is repeated since new roots may have appeared.
For example, if we are told, for the picture on the right, that only a and b are live, then the root d can be removed since d is dead. Then the rightmost node becomes a root, which also can be removed (since c is dead).
The Use of Algebraic Identities

Some of these are quite clear. We can of course replace x+0 or 0+x by simply x. Similar Considerations apply to 1*x, x*1, x-0, and x/1.

Strength reduction

Another class of simplifications is strength reduction, where we replace one operation by a cheaper one. A simple example is replacing 2*x by x+x on architectures where addition is cheaper than multiplication. A more sophisticated strength reduction is applied by compilers that recognize induction variables (loop indices). Inside a for i from 1 to N loop, the expression 4*i can be strength reduced to j=j+4 and 2^i can be strength reduced to j=2*j (with suitable initializations of j just before the loop). Other uses of algebraic identities are possible; many require a careful reading of the language
reference manual to ensure their legality. For example, even though it might be advantageous to convert ((a + b) * f(x)) * a to ((a + b) * a) * f(x)
it is illegal in Fortran since the programmer's use of parentheses to specify the order of operations can not be violated.
Does

a = b + c

x = y + c + b + r

contain a common sub expression of b+c that need be evaluated only once?

The answer depends on whether the language permits the use of the associative and commutative law for addition. (Note that the associative law is invalid for floating point numbers.)

7.8 FLOW-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 L2

….

L1 : gotoL2

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
```

can be replaced by

```
if a < b goto L2
….

L1 : goto L2
```

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

L3:....................................................(1)
```

may be replaced by

```
if a<b goto L2
goto L3

…….

L3:...................................................(2)
```

While the number of instructions in (1) and (2) is the same, we sometimes skip the unconditional jump in (2), but never in (1).Thus (2) is superior to (1) in execution time

7.9 REGISTER ALLOCATION

Instructions involving register operands are usually shorter and faster than those involving operands in memory. Therefore, efficient utilization of register is particularly important in generating good code. The use of registers is often subdivided into two sub problems:

1. During register allocation, we select the set of variables that will reside in registers at a point in the program.
2. During a subsequent register assignment phase, we pick the specific register that a variable will reside in. Finding an optimal assignment of registers to variables is difficult, even with single register values. Mathematically, the problem is NP-complete. The problem is further complicated because the hardware and/or the operating system of the target machine may require that certain register usage conventions be observed.

Certain machines require register pairs (an even and next odd numbered register) for some operands and results. For example, in the IBM System/370 machines integer multiplication and integer division involve register pairs. The multiplication instruction is of the form M x, y where x, is the multiplicand, is the even register of an even/odd register pair.

The multiplicand value is taken from the odd register pair. The multiplier y is a single register. The product occupies the entire even/odd register pair.

The division instruction is of the form D x, y where the 64-bit dividend occupies an even/odd register pair whose even register is x; y represents the divisor. After division, the even register holds the remainder and the odd register the quotient. Now consider the two three address code sequences (a) and (b) in which the only difference is

the operator in the second statement. The shortest assembly sequence for (a) and (b) are given in(c). Ri stands for register i. L, ST and A stand for load, store and add respectively. The optimal choice for the register into which 'a' is to be loaded depends on what will ultimately happen to e.

t := a + b t := a + b

t := t * c t := t + c

t := t / d t := t / d

(a) (b)

Two three address code sequences

```
L R1, a     L R0, a
A R1, b     A R0, b M
R0,         c A R0, c
D R0, d         SRDA
R0, ST R1, t    D R0, d
                ST R1, t

(a)             (b)
```

7.10 CHOICE OF OF EVALUATION ORDER

The order in which computations are performed can affect the efficiency of the target code. Some computation orders require fewer registers to hold intermediate results than others. Picking a best order is another difficult, NP-complete problem. Initially, we shall avoid the problem by generating code for the three -address statements in the order in which they have been produced by the intermediate code generator.

7.11 APPROCHES TO CODE GENERATION

The most important criterion for a code generator is that it produce correct code. Correctness takes on special significance because of the number of special cases that code generator must face. Given the premium on correctness, designing a code generator so it can be easily implemented, tested, and maintained is an important design goal Reference Counting Garbage Collection The difficulty in garbage collection is not the actual process of collecting the garbage--it is the problem of finding the garbage in the first place. An object is

considered to be garbage when no references to that object exist. But how can we tell when no references to an object exist? A simple expedient is to keep track in each object of the total number of references to that object. That is, we add a special field to each object called

a reference count . The idea is that the reference count field is not accessible to the Java program. Instead, the reference count field is updated by the Java virtual machine itself.
Consider the statement
Object p = new Integer (57);
which creates a new instance of the Integer class. Only a single variable, p, refers to the object. Thus, its reference count should be one.

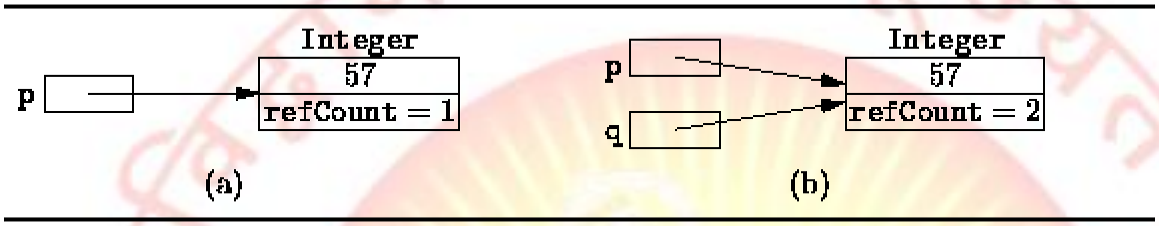


Figure: Objects with reference counters.

Now consider the following sequence of statements:

Object p = new Integer (57);
Object q = p;
This sequence creates a single Integer instance. Both p and q refer to the same object. Therefore, its reference count should be two.

In general, every time one reference variable is assigned to another, it may be necessary to update several reference counts. Suppose p and q are both reference variables. The assignment
p = q;

would be implemented by the Java virtual machine as follows:

```
if (p != q)
{
    if (p != null)
            --p.refCount;
    p = q;
    if (p != null)
            ++p.refCount;
}
```

For example suppose p and q are initialized as follows:

Object p = new Integer (57);
Object q = new Integer (99);
As shown in Figure ❑ (a), two Integer objects are created, each with a reference count of

one. Now, suppose we assign q to p using the code sequence given above. Figure ❑ (b) shows that after the assignment, both p and q refer to the same object--its reference count is two. And the reference count on Integer(57) has gone to zero which indicates that it is garbage.

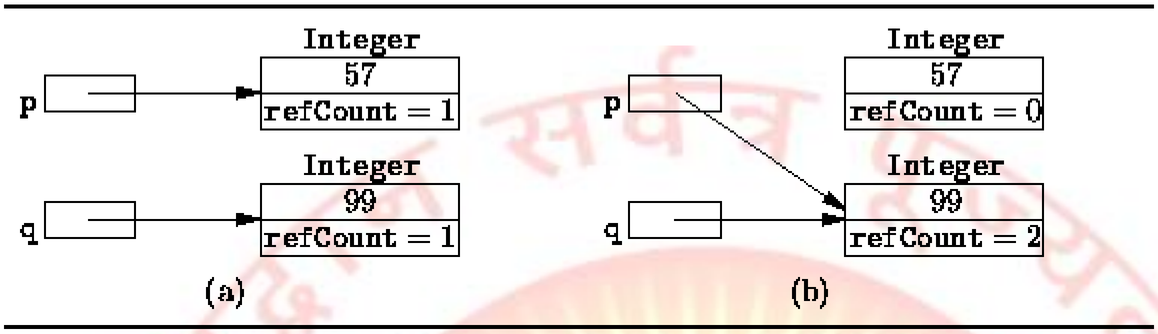


Figure: Reference counts before and after the assignment p = q.

The costs of using reference counts are twofold: First, every object requires the special reference count field. Typically, this means an extra word of storage must be allocated in each object. Second, every time one reference is assigned to another, the reference counts must be adjusted as above. This increases significantly the time taken by assignment statements.

The advantage of using reference counts is that garbage is easily identified. When it becomes necessary to reclaim the storage from unused objects, the garbage collector needs only to examine the reference count fields of all the objects that have been created by the program. If the reference count is zero, the object is garbage.

It is not necessary to wait until there is insufficient memory before initiating the garbage collection process. We can reclaim memory used by an object immediately when its reference goes to zero. Consider what happens if we implement the Java assignment p = q in the Java virtual machine as follows:

```
if (p != q)
{
   if (p != null)
         if (--p.refCount == 0)
            heap.release (p);
   p = q;
   if (p != null)
         ++p.refCount;
}
```

Notice that the release method is invoked immediately when the reference count of an object goes to zero, i.e., when it becomes garbage. In this way, garbage may be collected incrementally as it is created.

TEXT BOOKS:

1. Compilers, Principles Techniques and Tools- Alfred V Aho, Monical S Lam, Ravi Sethi, Jeffrey D. Ullman,2nd ed, Pearson,2007.
2. Principles of compiler design, V. Raghavan, 2nd ed, TMH, 2011.
3. Principles of compiler design, 2nd ed, Nandini Prasad, Elsevier

REFERENCE BOOKS:

1. http://www.nptel.iitm.ac.in/downloads/106108052/

2. Compiler construction, Principles and Practice, Kenneth C Louden, CENGAGE

3. Implementations of Compiler, A new approach to Compilers including the algebraic methods, Yunlinsu, SPRINGER