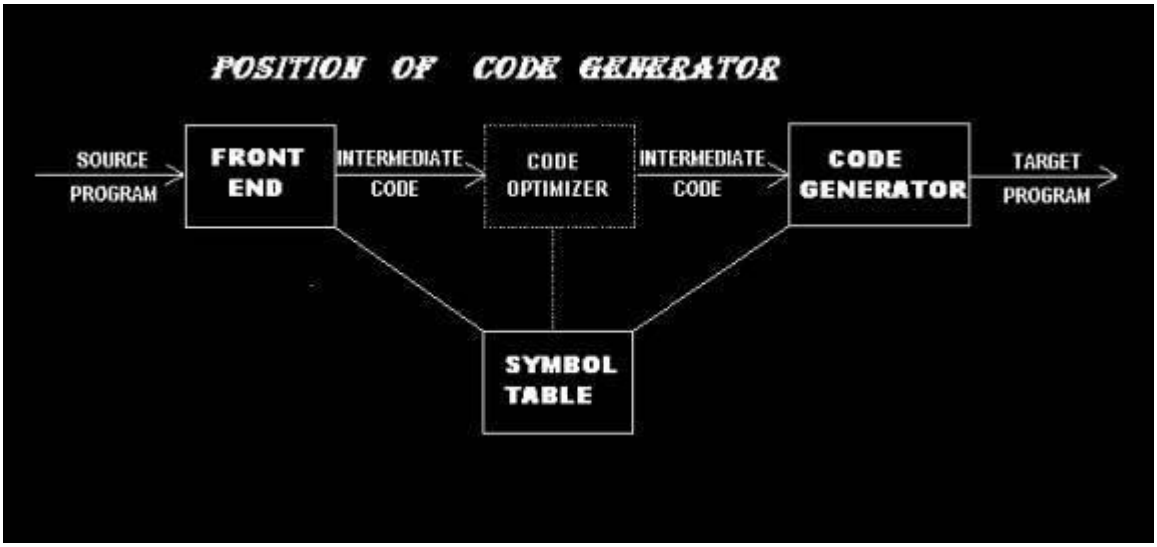
Unit-V

OBJECT CODE GENERATION:

The final phase in our compiler model is the code generator. It takes as input an intermediate representation of the source program and produces as output an equivalent target program.
The requirements traditionally imposed on a code generator are severe. The output code must be correct and of high quality, meaning that it should make effective use of the resources of the target machine. Moreover, the code generator itself should run efficiently.



ISSUES IN THE DESIGN OF A CODE GENERATOR

While the details are dependent on the target language and the operating system, issues such as memory management, instruction selection, register allocation, and evaluation order are inherent in almost all code generation problems.

INPUT TO THE CODE GENERATOR

The input to the code generator consists of the intermediate representation of the source program produced by the front end, together with information in the symbol table that is used to determine the run time addresses of the data objects denoted by the names in the intermediate representation.

There are several choices for the intermediate language, including: linear representations such as postfix notation, three address representations such as quadruples, virtual machine representations such as syntax trees and dags.

We assume that prior to code generation the front end has scanned, parsed, and translated the source program into a reasonably detailed intermediate representation, so the values of names appearing in the intermediate language can be represented by quantities that the target machine can directly manipulate (bits, integers, reals, pointers, etc.). We also assume that the necessary type checking has take place, so type conversion operators have been inserted wherever necessary and obvious semantic errors (e.g., attempting to index an array by a floating point number) have already been detected. The code generation phase can therefore proceed on the assumption that its

input is free of errors. In some compilers, this kind of semantic checking is done together with codegeneration.

TARGET PROGRAMS

The output of the code generator is the target program. The output may take on a variety of forms: absolute machine language, relocatable machine language, or assembly language.

Producing an absolute machine language program as output has the advantage that it can be placed in a location in memory and immediately executed. A small program can be compiled and executed quickly. A number of "student-job" compilers, such as WATFIV and PL/C, produce absolute code.

Producing a relocatable machine language program as output allows subprograms to be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader. Although we must pay the added expense of linking and loading if we produce relocatable object modules, we gain a great deal of flexibility in being able to compile subroutines separately and to call other previously compiled programs from an object module. If the target machine does not handle relocation automatically, the compiler must provide explicit relocation information to the loader to link the separately compiled program segments.

Producing an assembly language program as output makes the process of code generation somewhat easier .We can generate symbolic instructions and use the macro facilities of the assembler to help generate code .The price paid is the assembly step after code generation.

Because producing assembly code does not duplicate the entire task of the assembler, this choice is another reasonable alternative, especially for a machine with a small memory, where a compiler must uses several passes.

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:

Invoke a function getreg to determine the location L where the result of the computation y op z should be stored.

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.

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.

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:

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, R_0 SUB b, R0	R_0 contains t	t in R_0
u : = a – c	MOV a , R1 SUB c , R1	R_0 contains t R1 contains u	t in R_0 u in R1
v : = t + u	ADD R_1, R_0	R_0 contains v R_1 contains u	u in R_1 v in R_0

d : = v + u	ADD R_1, R_0
	
	MOV R_0, d
		R
		0 contains d
			d in R_0
			d in R_0 and memory

Generating Code for Indexed Assignments

The table shows the code sequences generated for the indexed assignment statements a : = b [i] and a [i] : = b

Statements	Code Generated	Cost
a : = b[i]	MOV b(R_i), R	2
a[i] : = b	MOV b, a(R_i)	3

Generating Code for Pointer Assignments

The table shows the code sequences generated for the pointer assignments a : = *p and *p : = a

Statements	Code Generated	Cost
a : = *p	MOV *R_p, a	2
*p : = a	MOV a, *R_p	2

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 theprogram.

2. During a subsequent **register assignment** phase, we pick the specific register that a variable will residein.

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 theform 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) fig. 2 Two three address codesequences

L	**R1, a**	**L**	**R0, a**		
A	**R1, b**	**A**	**R0, b**		
M	**R0, c**	**A**	**R0, c**		
D	**R0, d**	**SRDA**	**R0, 32**		
ST	**R1, t**	**D**	**R0, d ST**	**R1,t**	
(a)		**(b)**			

THE DAG REPRESENTATION FOR BASIC BLOCKS

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

Leaves are labeled by unique identifiers, either variable names or constants.

Interior nodes are labeled by an operator symbol.

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

Input: A basic block

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

A label for each node. For leaves, the label is an identifier. For interior nodes, an operator symbol.

For each node a list of attached identifiers to hold the computed values.

Case (i) x : = y OP z

Case (ii) x : = OP y

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).

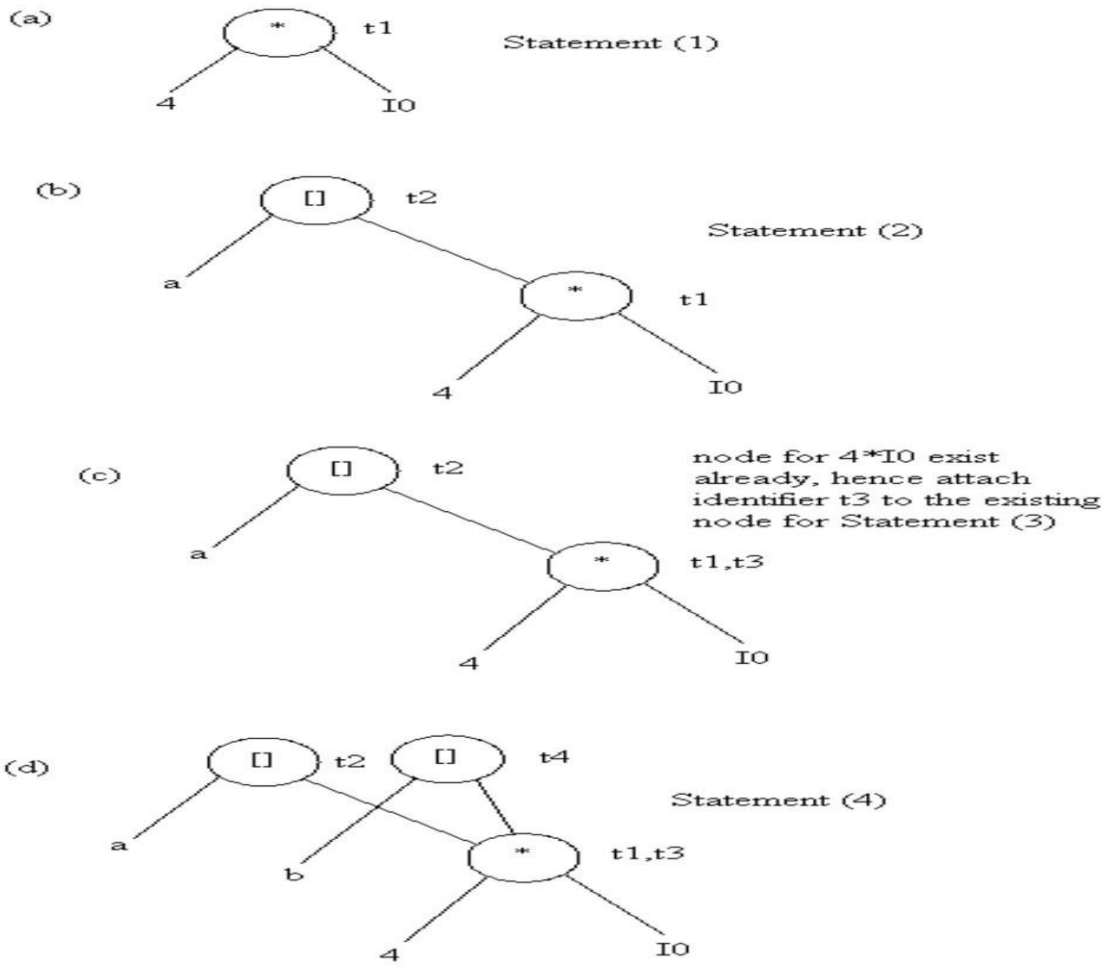
Step 3: Delete x from the list of identifiers for node(x). Append x to the listof attached identifiers for the

node n found in step 2 and set node(x) to n.

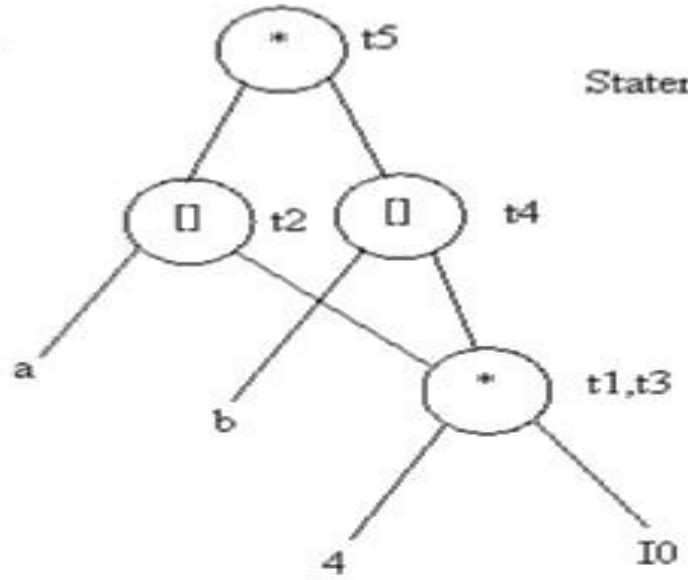
Example: Consider the block of three- address statements:

t_1 := 4* i
t_2 := a[t_1]
t_3 := 4* i
t_4 := b[t_3]
t_5 := t_2*t_4
t_6 := prod+t_5
prod := t_6
t_7 := i+1
i := t_7
if i<=20 goto (1)

Stages in DAG Construction

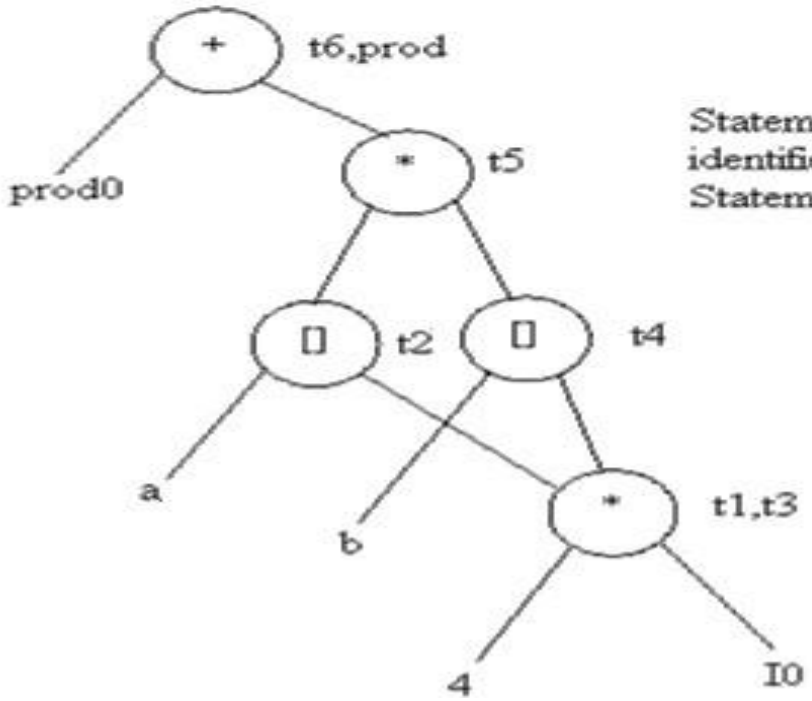


(e)



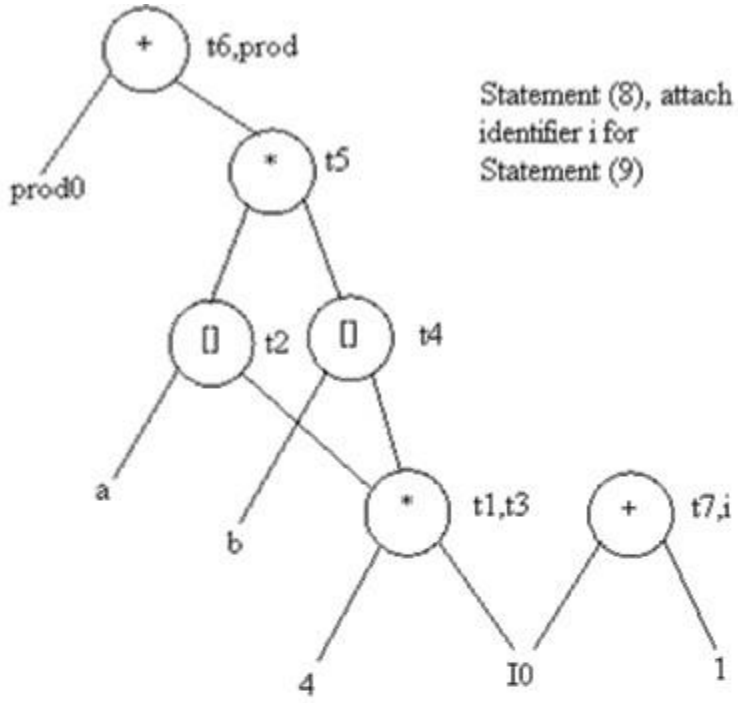
Statement (5)

(f)



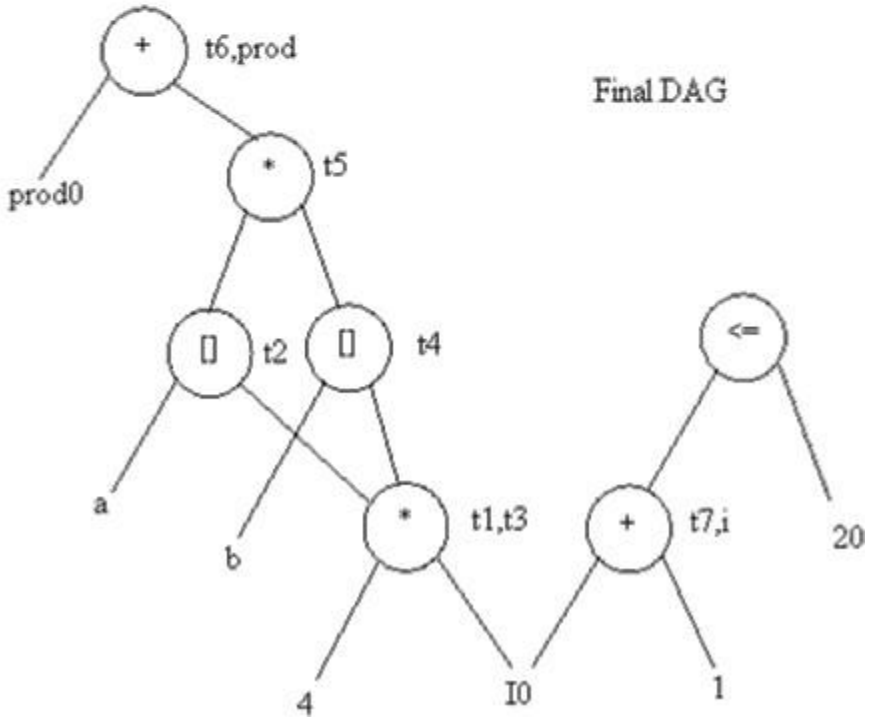
Statement (6), attach identifier prod for Statement (7)

(g)



Statement (8), attach identifier i for Statement (9)

(h)



Final DAG

GENERATING CODE FROM DAGs

The advantage of generating code for a basic block from its dag representation is that, from a dag we can easily see how to rearrange the order of the final computation sequence than we can starting from a linear sequence of three-address statements or quadruples.

Rearranging the order

The order in which computations are done can affect the cost of resulting object code.

For example, consider the following basic block:

$t_1 := a + b$
$t_2 := c + d$
$t_3 := e - t_2$
$t_4 := t_1 - t_3$

Generated code sequence for basic block:

MOV a , R_0
ADD b , R_0
MOV c , R_1
ADD d , R_1
MOV R_0 , t_1
MOV e , R_0
SUB R_1 , R_0
MOV t_1 , R_1
SUB R_0 , R_1
MOV R_1 , t_4

Rearranged basic block:

Now t1 occurs immediately before t4.

$t_2 := c + d$
$t_3 := e - t_2$
$t_1 := a + b$
$t_4 := t_1 - t_3$

Revised code sequence:

MOV c , R_0
ADD d , R_0
MOV a , R_0
SUB R_0 , R_1
MOV a , R_0
ADD b , R_0
SUB R_1 , R_0
MOV R_0 , t_4

In this order, two instructions MOV R_0 , t_1 and MOV t_1 , R_1 have been saved.