12

CODE GENERATION

12.1 INTRODUCTION TO CODE GENERATION

- → It is the final phase in the process of compilation.
- → It takes intermediate code as an input and generates target machine code as output.
- → This is the phase of compiler that generates the object code from the sequence of optimized three address codes as shown in fig 12.1.

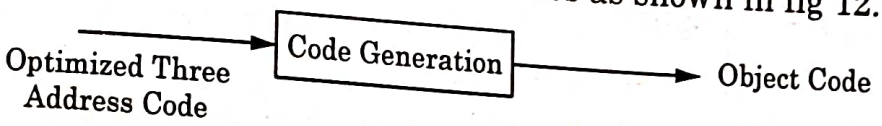


Fig. 12.1 : Generation of Object Code

12.2 OBJECT PROGRAMS

- → **Input to the code generation phase** is an intermediate language program *i.e.* Quadruples. Triples, Three Address code and Postfix Notation.
- → **Output of Code generation** is an object code or machine code. Object code can of following types :–
  1. **Absolute code**
  2. **Relocatable machine code**
  3. **Assembly code**

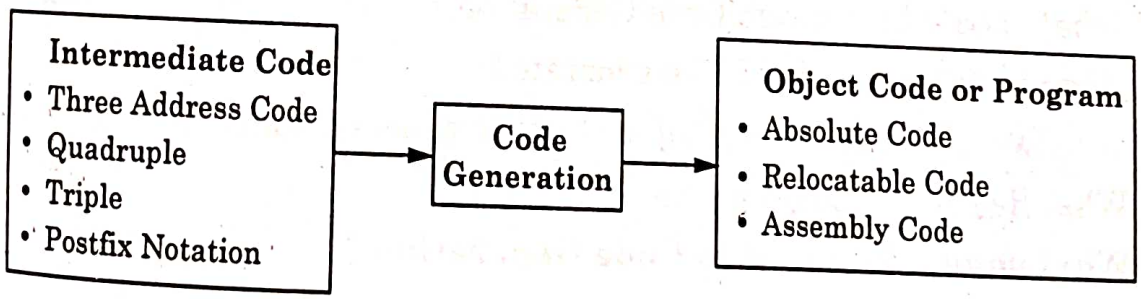


Fig. 12.2 : Types of Object Programs

12.2.1 Types of Object Programs

1. Absolute Code : It is a machine code that contains reference to actual address with in program's address space.

Advantages of absolute code

(*a*) The generated code can be placed directly in the memory.

Application : A number of "student - job" compilers such as WATFIV and PL/C produce absolute code.

2. Relocatable 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 with the help of a linking loader.

Advantage of Relocatable Code

(*a*) We gain a great deal of flexibility in being able to compile subroutines separately.

Application. Many commercial compiler produce relocatable object modules.

3. Assembly Code : Generation of an assembly language program as output makes the process of code generation easier.

Advantage of assembly code : We can generate symbolic instructions and use the macro facilities of the assembler to help in generation of code.

Application : Machines with a small memory use assembly code.

12.3 PROBLEMS IN CODE GENERATION

- → Before performing code generation, we have to assume that source language has been scanned, parsed and translated into intermediate code.
- → Code Generation appears an easy task but there are some problems or issues to be concerned while generating code.

Following are the main problems in Code Generation

1. What should be input to Code Generation ?
2. What instructions should we generate ?
3. What should be order of computations to generate efficient code ?
4. What Registers should we use ?

1. What should be input to Code Generation ?

- → The code generation phase takes intermediate code as input.
- → There are several choices for intermediate codes

(a) Linear representation *i.e.* Postfix Notation
(b) Three address code *i.e.* Quadruples, Triples
(c) Graphical Representation *i.e.* Syntax Trees, DAGs

2. **What Instructions should we generate ?**
   - → Sometimes, we have different machine code sequences for a given three address statement. Deciding which of them will be best is a difficult task.
   - → For example, For a three Address Statement A = A + 1, we can implement it in different ways.
     (a) By "add-one-to storage Instruction" (AOS).
     ```
     AOS   A
     ```
     (b) By simple Assembly Language
     ```
     LOAD   A
     ADD    # 1
     STORE  A
     ```

3. **What should be order of computations to generate efficient code ?**
   - → Mainly, the code is generated for three address statements in the order in which they have been produced by previous phases.
   - → But, sometimes if order of computation is changed, then computation will require less registers to hold intermediate results. So, it is a great problem to decide the order of computation.

4. **What Registers should we use ?**
   The use of Registers is subdivided into two sub-problems :–
   (a) **Register Allocation :** Deciding appropriate variable that will reside in registers.
   (b) **Register Assignment :** Deciding specific register in which to store a particular variable.

Some machines require register-pairs for some operands and results.
For examples :–
IBM system use register pairs for Integer multiplication and Integer division.
(i) Multiplication instruction is of form

M	X	Y

where X = multiplicand which refers to even register of an even/odd register pair.
Y = multiplier

(ii) Division instruction is form

D	X	Y

where X = dividend which refers to even register
Y = divisor

After division, Remainder will be saved in even register and Quotient will be saved in odd register.

12.4 MACHINE MODEL

For designing good code generator, it is necessary to have prior knowledge of target machine.

Consider following assumptions for a sample machine.

- A machine consist of 2^{16} bytes of memory.
- There are 8 general purpose registers R_0, R_1 ..... R_7.
- Each register can hold 16 bit data.
- Two address instructions are of form

op Source, destination

where op is an opcode and source, destinations are data fields.

- Following op-codes are used

MOV : Moves from source to destination
ADD : add source to destination
SUB : Subtracts source from destination
Source and destinations are specified by registers and memory locations.

12.4.1 Addressing Modes

Following are the various addressing modes used in various instructions :-

Addressing Mode	Form	Address	Description
absolute	M	M	address of memory Loc. M.
register	R	R	Register R contains operands
indexed	C (R)	C + contents (R)	Value of C is added to contents of register R to produce address of operand.
indirect register	* R	contents (R)	Register R contains address of operand.
indirect indexed	* C (R)	contents (C + contents (R))	Value C is added to contents of register R to produce address of word containing address of operand.
Literal or Immediate	# C	C	Word following instruction contain literal C

Fig. 12.3 : Addressing Modes

Examples of Instruction for various addressing modes

1. **absolute or Register Mode**

   e.g. MOV R0, M

   It stores contents of register R0 into memory location M.

2. **Indexed Addressing Mode**
   MOV 7 (R1), M
   It stores value (7 + contents (R1) to Memory Location M.
3. **Indirect Indexed Mode**
   MOV * 7 (R0), M
   It stores value contents (7 + contents (R0)) to memory location M.
4. **Literal Addressing Mode**
   MOV # 7, R0
   It stores constant 7 into register R0.

12.4.2 Cost of the Instruction

Cost of Instruction can be computed as one plus the cost associated with source and destination addressing modes.

Cost of Addressing Modes

1. **Absolute or Memory Address** add **cost 1** to Instruction.
2. **register (R)** and **indirect Register (* R)** add **no cost** to Instruction.
3. **Indexed (C (R)), indirect indexed (* C (R)), Literal (# C)** add **cost 1** to instruction.

Example 1. *Compute cost for following instructions*

MOV a, R0
ADD b, R0
MOV R0, c

Ans.

MOV *a*, R0	Cost of memory var '*a*' + Cost of Register R0 + 1	1 + 0 + 1 = 2
ADD *b*, R0	Cost of memory var '*b*' + Cost of Register R0 + 1	1 + 0 + 1 = 2
MOV R0, *c*	Cost of Register R0 + cost of memory var. 'c' + 1	0 + 1 + 1 = 2
		Total Cost = 6

Example 2. *Compute cost for*

*MOV * R1, * R0*
*ADD * R2, * R0*

2. **Indexed Addressing Mode**
   MOV 7 (R1), M
   It stores value (7 + contents (R1) to Memory Location M.
3. **Indirect Indexed Mode**
   MOV * 7 (R0), M
   It stores value contents (7 + contents (R0)) to memory location M.
4. **Literal Addressing Mode**
   MOV # 7, R0
   It stores constant 7 into register R0.

12.4.2 Cost of the Instruction

Cost of Instruction can be computed as one plus the cost associated with source and destination addressing modes.

Cost of Addressing Modes

1. **Absolute or Memory Address** add **cost 1** to Instruction.
2. **register (R)** and **indirect Register (* R)** add **no cost** to Instruction.
3. **Indexed (C (R)), indirect indexed (* C (R)), Literal (# C)** add **cost 1** to instruction.

Example 1. *Compute cost for following instructions*

MOV a, R0
ADD b, R0
MOV R0, c

Ans.

MOV *a*, R0	Cost of memory var '*a*' + Cost of Register R0 + 1	1 + 0 + 1 = 2
ADD *b*, R0	Cost of memory var '*b*' + Cost of Register R0 + 1	1 + 0 + 1 = 2
MOV R0, *c*	Cost of Register R0 + cost of memory var. 'c' + 1	0 + 1 + 1 = 2
		Total Cost = 6

Example 2. *Compute cost for*

*MOV * R1, * R0*
*ADD * R2, * R0*

Ans.

MOV * R1, * R0	cost of indirect Register (* R1) + cost of indirect Register (* R0) + 1	0 + 0 + 1 = 1
ADD * R2, * R0	Cost of indirect Register (* R2) + cost of indirect Register (* R0) + 1	0 + 0 + 1 = 1
		Total Cost = 2

Example 3. *Compute cost for*

*SUB 7 (R0), * 10 (R1)*

Ans. Cost of indexed (7) R0 + cost of indirect indexed (* 10 (R1)) + 1

$\therefore$ 1 + 1 + 1 = 3

12.5 SIMPLE CODE GENERATOR

The code is generated from the given three address statement (Quadruples). Some Assumptions are taken before generating code.

Assumptions :–

1. For each operator in a quadruples, there is corresponding machine code operator.
2. Computed Results can be left in registers if their register is needed for another computation or before Procedure call, Jump, or Labeled Statement *i.e.* everything must be stored just before end of basic block.
3. Code generation algorithm stores everything when moving across basic block boundaries as well as when procedure calls are made.

12.5.1 Next-use Information

The Next-use Information is used to identify the list of all names that are used in subsequent blocks of the program. If name in a register is no longer used after a particular statement, it can be used to store other names.

Use of a Name

Consider the statements

Statement 1: $a = b$

Statement 2: $c = a + d$

Control flows from statement 1 to statement 2. Statement 2 is said to use value of 'a' computed at statement 1 if there is no intermediate statements computing and assigning the value to a between statement 1 and statement 2.

Backward Scan to Compute Next-Use Information

In each statement or quadruple of form

i : $a = b$ **op** c in the basic block where i is the Label, op is any operator.

Perform following steps :-

1. Information currently found in symbol table regarding next use & Liveness of a, b and c is attached to statement i.
2. Set a to "not Live" and "no next use" in Symbol table.
3. Set b and c to "live" and next use of b and c to i in the symbol table.

12.5.2 Code Generation Algorithm

For a given three Address statement (intermediate code), we can generate target code, using some sequence of steps which is called **code generation algorithm.**

Code generation algorithm uses two descriptors to keep track of what is currently in register and memory addresses.

1. **Register Descriptor :** It is to keep track of what is currently in each register. Initially it is empty. During code generation of statements, it stores value of computations.

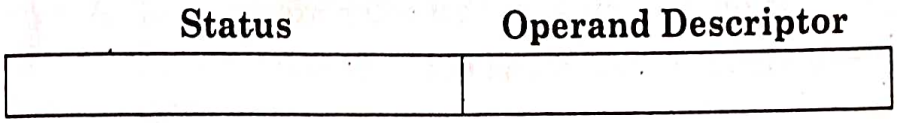


Fig 12.4 : Register Descriptor

Status : field is boolean type. It tells whether register is occupied or not.

Operand descriptors : It contains pointer to operand descriptor having latest value in register.

2. **Address Descriptor :** It stores the location where current value of name can be found at run time.

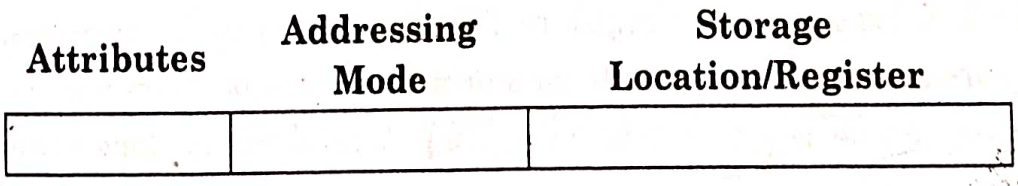


Fig 12.5 : Address Descriptor

Attributes : means type of operand.

Addressing modes : can be Absolute, Register, indirect etc.

Storage Location / Register : Indicates whether address is in storage location or in register

→ **getreg ( ) :** It is a function that returns the location L to hold value of A for assignment A = B op C.

Algorithm :

Input : Sequence of three-address statements of form A = B op C.

Output : Target code

Method :

1. Invoke the function **'getreg ( )'** to determine location L where result of B op C should be stored. L can be register or memory location.
2. Consult Address descriptor for B to determine B′ (current Location of B). Prefer register for B′ if B is currently both in memory and register. Move current value B to L by **MOV B′, L.**
3. Generate instruction **op C′, L** where C′ is current location of C. Update address descriptor of A to indicate that A is in Location L. If L is a register, update its descriptor to indicate that it will contain at run time the value of A.
4. If current value of B and C have no next uses and are in registers, alter the register descriptor to indicate that after execution of A = B op C, these registers will not contain B and C.

12.6 REGISTER ALLOCATION AND ASSIGNMENT

→ **Register Allocation :** Deciding what names in a program should reside in registers.

→ **Register Assignment :** Deciding in which register, names should reside or stored.

The most commonly used strategy to register allocation and assignment is to assign specific values to specific registers. For example for base addresses, separate set of registers can be used. Similarly for storing the stack pointers again a separate set of registers can be assigned, arithmetic computations can be done using separate set of registers.

→ **Advantage :** The design process of compiler for code generation becomes simple.

→ **Disadvantage :** The design of compiler becomes complicated because of restrictive use of registers. At the same time some registers remain totally unused and some registers get overloaded.

The strategies used in Register Allocation and Assignment are :-

1. Global register allocation
2. Usage count
3. Register assignment for outer loop

12.6.1 Global Register Allocation

During code generation registers are used to hold the values for the duration of single block. All the live variables are stored at the end of each block. For the variables that are used consistently we can allocate specific set of registers. Hence allocation of variables to specific registers that is consistent across the block boundaries is called global register allocation.

Strategies adopted for global register allocation :-

→ The global register stores the most frequently used variables in fixed registers throughout the loop.

→ Assign some fixed number of global registers to hold the most active values in each inner loop.

12.6.2 Usage Counts

→ The **usage count** is the count for use of some variable x in some register used in any basic block.

→ Usage count gives an approximation about how many units of cost can be saved by selecting a specific variable for global register allocation.

→ The approximate formula for usage count for Loop L in some basic block B is given by

$$\sum_{\text{Block B in L}} \text{use}(x, B) + 2 * \text{live}(x, B)$$

where **use** (x, B) is no. of times x is used in block B before any definition of x.

live $(x, B) = 1$ if x is live on exit from B otherwise live $(x, B) = 0$.

For example :

Consider blocks B1, B2, B3, B4 and count the usage count a, b, c, d, e, f for block B in the following Loop of flow graph (Fig. 12.6).

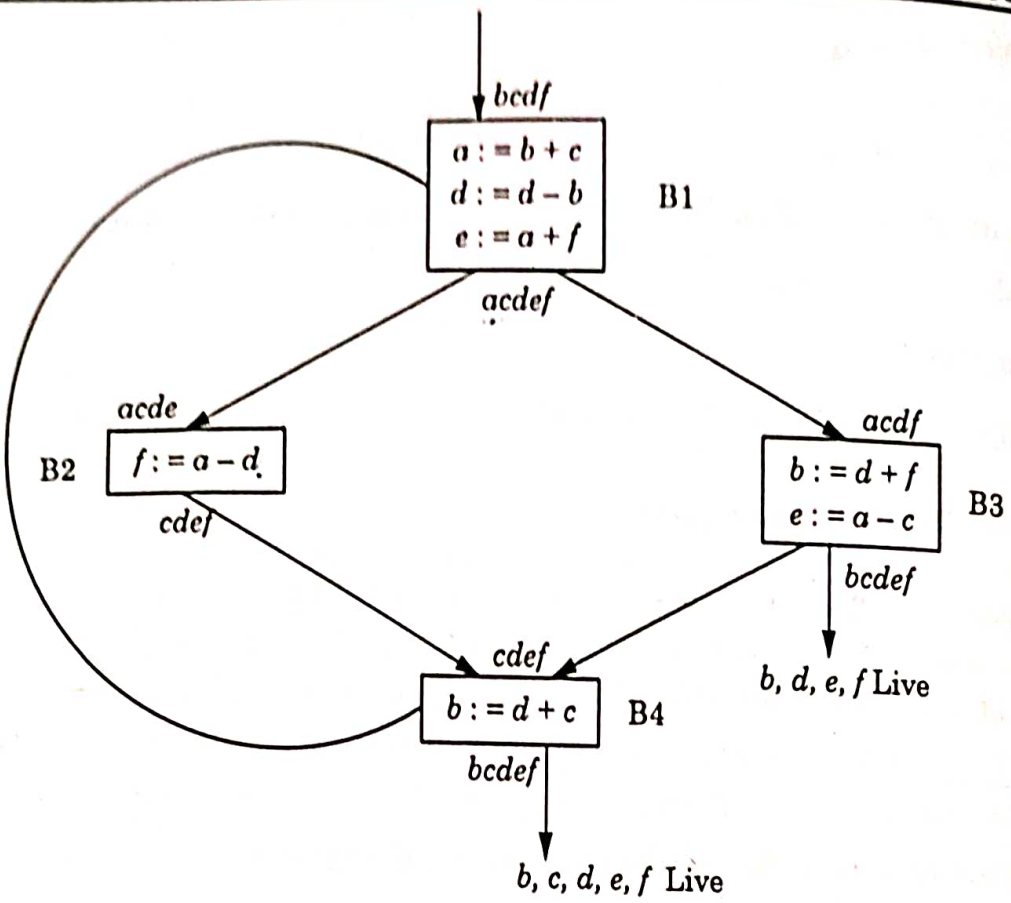


Fig. 12.6 : Flow Graph

For Block B1

Usage Count for Block B1 for variable *a*

∵ *a* is defined in B1 before use

∴ use (*a*, B1) = 0

∵ *a* is Live on exit of B1

∴ live (*a*, B1) = 1

∴ [use (*a*, B1) + 2 * live (*a*, B1)] = 0 + 2 * 1 = 2 ...(1)

For Block B2

Usage Count for Block B2 for variable *a*

∵ *a* is used in B2 before definition

∴ use (*a*, B2) = 1

∵ *a* is not live on exit from B2

∴ live (*a*, B2) = 0

∴ [use (*a*, B2) + 2 * live (*a*, B2)] = 1 + 2 * 0 = 1 ...(2)

For Block B3

Usage Count for Block B3 for variable *a*

use $(a, B3) = 1$ ∵ a is used before def. in B3

live $(a, B3) = 0$ ∵ a is not live on exit from B3

∴ [use $(a, B3) + 2 * \text{live}\ (a, B3)] = 1 + 2 * 0 = 1$ ...(3)

For Block B4

use $(a, B4) = 0$ ∵ a is not used in Block B4.

live $(a, B4) = 0$ ∵ a is not live on exit from B4.

∴ [use $(a, B4) + 2 * \text{live}\ (a, B4)] = 0 + 2 * 0 = 0$ ...(4)

∴ **Usage count for variable a in Loop**

∴ $= \sum_{B \text{ in } L} \text{use}\ (a, B) + 2 * \text{live}\ (a, B)$

= Statement 1 + statement 2 + statement 3 + statement 4

= 2 +1 + 1 + 0 = 4

∴ usage count of $a = 4$

∴ Compiler can save 4 units of cost by selecting a for global register allocation

i.e. If a is assigned to any global register, than 4 units of cost can be saved.

Similarly, usage count of b, c, d, e, f can be computed as 5, 3, 6, 4 and 4 respectively.

- More the number of usage count, more the saving on assigning that particular variable to register.
- Since, d has highest cost saving of 6. It will be allocated to register first.

12.6.3 Register Assignment for Outer Loops

If there are two loops L1 (outer loop) and L2 (inner loop).

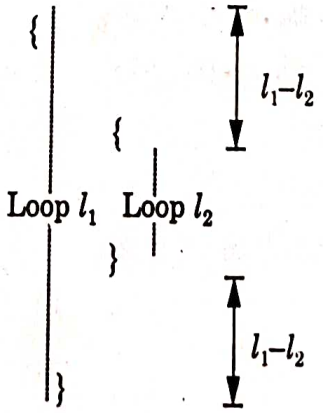


Fig 12.7 : Assignment for outer Loop

Allocation of a in Loop can be done as follows :-

(a) If a is allocated register in loop l_2, then it should not be allocated in $l_1 - l_2$.

(b) If a is allocated register in l_2 and not in l_1 then load a on entrance to l_2 and store a on exit from l_2.

(c) If a is allocated register in loop l_1 but not in l_2, then store a on entrance to l_2 and load a on entrance of $l_1 - l_2$.

12.7 CODE GENERATION FROM DAG'S

In earlier sections, we have converted three address code to target code. But, we can also generate code from DAG in a very simple manner.

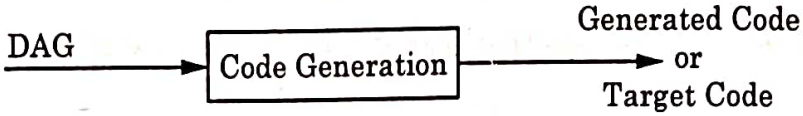


Fig 12.8 : Code generation from DAG

Algorithms used to generate code from DAG :-

1. Rearranging order
2. Heuristic ordering
3. Labeling algorithm

12.7.1 Rearranging Order

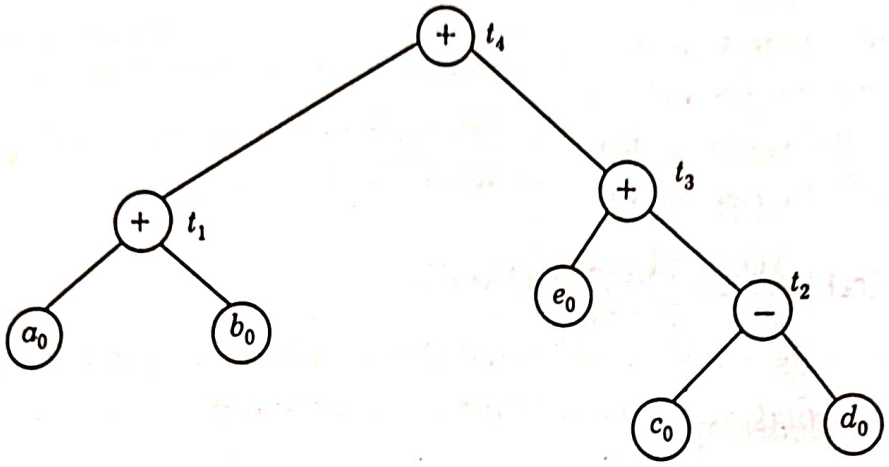
- Sometimes, order of evaluation of three address code can affect cost of object code or target code.
- By changing order of computation, we can minimize the cost of object code.

For example :

Consider the expression $(a + b) + (e + (c - d))$. Its **three address code** will be

$$
\begin{aligned}
t1 &:= a + b \\
t2 &:= c - d \\
t3 &:= e + t2 \\
t4 &:= t1 + t3
\end{aligned}
$$

Corresponding DAG for above three address code



Object code for above three address code can be generated line by line :

MOV *a*, R0	
ADD *b*, R0	
MOV *c*, R1	
SUB *d*, R1	R1 contain *c*-*d*
MOV R0, *t*1	t1 = *a* + *b*
MOV *e*, R0	
ADD R1, R0	Ro contains *e* + (*c* – *d*)
MOV *t*1, R1	R1 contains *a* + *b*
ADD R0, R1	
MOV R1, *t*4	*t*4 = (*a* + *b*) + (*e* + (*c* – *d*))

On changing the ordering sequence of three address code statements :–

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

} computation of $t1$ occurs immediately before $t4$

∴ We get revised object code having less number of instructions *i.e.* more cient code.

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

Saving of two Instructions
MOV R0, $t1$
MOV $t1$, R1

12.7.2 Heuristic Ordering

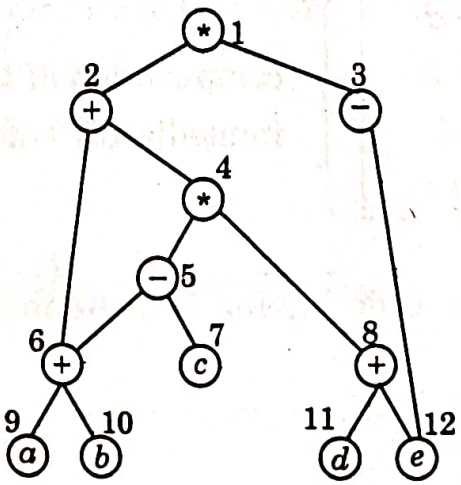
→ It lists the nodes of a DAG such that the node's reverse listing gives the order of computation.

Algorithm :
Input : DAG
Output : Ordering of nodes which evaluates order of computation.
Method :

1. While (there exists an unlisted interior node)
   {
2. Select an unlisted node n, whose parent have been listed
3. List n
4. While (Leftmost child m of n has no unlisted parents and m is not leaf)
   {
5. list m
6. $n = m$
   }
   }

Fig 12.9 : Node Listing Algorithm

Consider the DAG



→ Mark the numbering from top to bottom and left to right on DAG *i.e.* from 1 to 12.

→ Initially, **unlisted interior** nodes are
1, 2, 3, 4, 5, 6, 8

→ Select unlisted node 1 with no unlisted parent
∴ $n = 1$; List 1

→ The Leftmost child of 1 is 2 and parent of 2 is 1 which is listed.
∴ List 2 and set $n = 2$.

→ Find the Leftmost child of 2 and which is 6. But 6 has unlisted parent 5.
∴ we cannot select 6.
∴ Go to node 3.

→ Select unlisted node 3, which has parent 1 which is listed.
∴ $n = 3$, List 3

→ The Leftmost child of 3 is 4 and Parent of 4 is 3 which is Listed.
∴ List 4 and set $n = 4$.

→ Find Leftmost child of 4 and which is 5.
Now 5 has Listed parent 4.
∴ List 5

→ Find Leftmost child of 5 which is 6.
Now 6 has Listed parent 2 and 4.
∴ List 6

→ Now 8 is only remaining interior node.
So, List 8

→ ∴ Resulting List is 1 2 3 4 5 6 8

→ Reverse List or order of Evaluation will be
8 6 5 4 3 2 1

8 ⟶ $t8 = d + e$
6 ⟶ $t6 = a + b$
5 ⟶ $t5 = t6 - c$
4 ⟶ $t4 = t5 * t8$
3 ⟶ $t3 = t4 - e$
2 ⟶ $t2 = t6 + t4$
1 ⟶ $t1 = t2 * t3$

It gives the best three address code which can be further converted to object code.

12.7.3 Labeling Algorithm

→ This algorithm takes as it input the tree representation of the three address code.

→ **Algorithm has two steps :-**

1. Label each node of tree bottom up with an integer number. The number shows the fewest number of registers required to evaluate tree.
2. Traversal of tree in the order computed by labels. So, traversal gives the generated code. *i.e.* generation of code is done from labeled tree.

12.7.3.1 Labeling each node

→ Start from the bottom of tree and assign

Label of each left leaf = 1

Label of each right Leaf = 0

→ If l = label of left child of node n

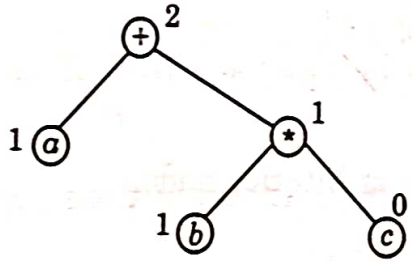
r = label of right child of node n

∴ Label of each node n

$$\text{label}(n) = \begin{cases} \max(l, r) & \text{if } l \neq r \\ l+1 & \text{if } l = r \end{cases}$$

For example,

Labeling of tree for expression $a + b * c$



Algorithm :-

```
if (n is leaf)
{
    If n is Leftmost child of its parent
        {
            label (n) = 1
        }
    else
        {
            label (n) = 0
        }
}
else if (n is interior node)
{
    If n1 , n2 , ..... nk are children of n
    such that label (n1) ≥ label (n2) ≥ ..... ≥ label (nk) then
    label (n) = max [label (ni) + i – 1]
                1≤i≤k
```

Fig 12.10 : Algorithm for Assigning Labels to Nodes.

12.7.3.2 Generation of code from Labeled tree

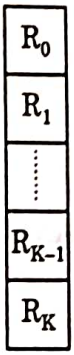
Algorithm :

Input : Labeled tree T

Output : Machine code

Assumptions :

- T has only binary operators.
- Procedure **gencode (*n*)** produce object code for node labeled *n*.
- **Register-stack** consist of all registers R_0, R_1, ... R_k



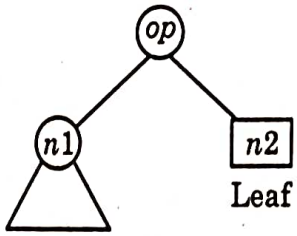
- **gencode (*n*)** evaluates *n* in register which is on top of stack.

Method :

Case 1 : If *n* is leaf and is leftmost child of its parent

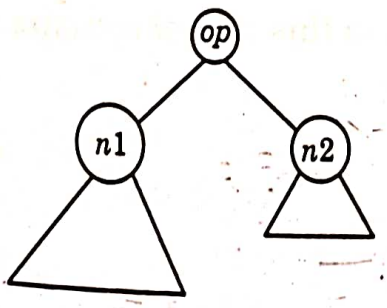
Leaf node  → Generate Load instructions *i.e.* Load label (*n*) into register. *i.e.* gencode (*n*).

Case 2 : If Right child is leaf & left child is subtree



- → *n*1 is first evaluated in the register R which is pushed at top of Register-Stack.
- → gencode (n1);
- → Instruction op *n*2, R will be executed.

Case 3 : If Left child *n*1 requires more number of registers than right hild *n*2.

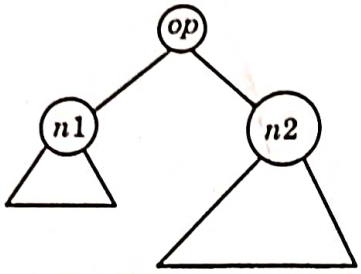


- → Evaluate Left Subtree first then the right subtree.
- → ∴ order of execution will be

gencode (*n*1);

gencode (*n*2);

Case 4 : If Right child $n2$ requires more registers than $n1$



- Firstly, swap top two registers in Register-stack.Then order of execution will be

  gencode ($n2$) ;

  gencode ($n1$);
- Evaluate right subtree into register and store register just below top of stack Register-stack. Then Evaluate left subtree into register which is on top of stack.
- Again swap top registers which make Register-stack as it was earlier.

12.8 PEEPHOLE OPTIMIZATION

- **Peephole optimization** is an effective technique for locally improving the target code. Short sequences of target code instructions are examined and are replaced by faster sequences.
- **Peephole** means a **small window** is moved over target code to apply some transformations.
- Each improvement may generate opportunities for additional improvements.
- Repeated passes of improvement are applied on code to get maximum benefit.

Various types of Peephole Optimization are :-

1. Redundant instruction Elimination
2. Removal of Unreachable Code
3. Multiple Jumps Elimination
4. Algebraic Simplification
5. Reduction in Strength
6. Use of Machine Idioms

1. Redundant Instruction Elimination

Redundant Loads and stores can be eliminated in this type of optimization.

For example

```
MOV R0, x
MOV x, R0
```

Second instruction can eliminated, since x is already in R0.

2. Removal of Unreachable Code

We can eliminate the unreachable instructions from a program to speedup execution.

For example :–

```
Sum = 0
If (Sum)
    printf (" % d", Sum) ;
```

This if statement will never get executed & hence, we can eliminate such unreachable code.

An unlabeled instruction that immediately follows an unconditional jump can possibly be removed.

For example :–

```
# define debug 0
    :
if (debug)
{ print debugging information }
```

→ Above code can be changed to intermediate code & then two goto's can be removed by single goto to get optimized code.

```
If debug = 1 goto L1
goto L2
L1 : print debugging information
L2 :
```

⇒

```
If debug ≠ 1 goto L2
print debugging information
L2 :
```

Intermediate code **Optimized Code**

In optimized code, if debug is set to 0 at beginning of program. Its value will be substituted in if-statement & will give.

```
If 0 ≠ 1 goto L2
  print debugging information
L2 :
```

Since, If statement will be always true

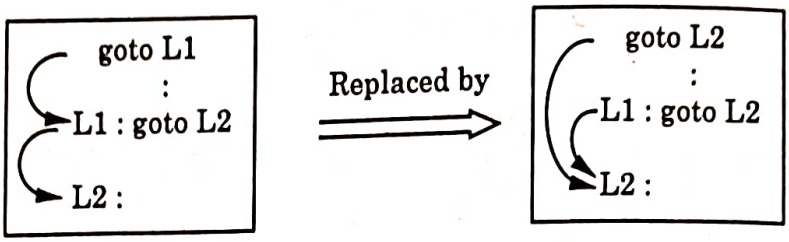
Therefore, it will always jump to L2.

∴ Debugging information will never execute and so can be eliminated.

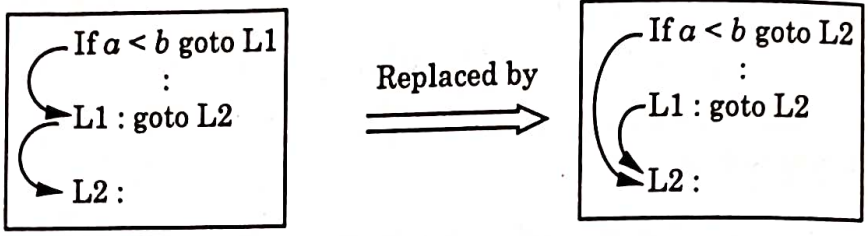
3. Multiple Jumps Elimination

Sometimes, there can be jumps to jumps *i.e.* either conditional jumps to jumps or jumps to unconditional jumps which can be removed.

For example :-



Another example :



4. Algebric Simplification

The statements such as

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

can be eliminated by peephole optimization. Because adding with 0 or multiplying with 1 will not make any difference in value of x.

5. Reduction in Strength : Some machine instructions are cheaper than other. To improve performance of the intermediate code, we can replace these instruction by equivalent cheaper instruction.

For example :

(*a*) x^2 is cheaper than $x * x$.

(*b*) Addition and Subtraction is cheaper than multiplication and division.

So, we can replace multiplication, division by equivalent addition and subtraction.

6. Machine Idioms :

- Some machines have hardware instructions to implement specific operations efficiently.
- We can replace the target instructions into equivalent hardware instructions to improve efficiency.
- For example, some machines have auto-increment or auto-decrement addressing modes to perform add or subtract operation (*i.e.* A = A + 1 or A = A – 1).
- Use of auto-increment or auto-decrement instead of instructions LOAD, ADD, STORE improve the performance of code.

KEY POINTS TO REMEMBER

⇒ **Code generation :** Final phase of compilation. It converts intermediate code *i.e.* three address code into target code or object code.

⇒ **Object Program :** It is the output of code generator

Types :

(*a*) Absolute code (*b*) Relocatable Machine Code (*c*) Assembly code

⇒ **Problems in Code generation**

(*a*) Input to code generator

(*b*) Instruction to be generated

(*c*) Order of computations to generate efficient code

(*d*) Registers to be used.

⇒ **Machine Model**

A machine having 2^{16} bytes of memory & general purpose registers R_0, R_1 ..... R_7 with addressing nodes like absolute, register, indexed, indirect register, indirect indexed, Literal.

⇒ **Register Descriptor :** Keeps current value stored in each register.

⇒ **Address Descriptor :** Stores the Location where current value is stored.

⇒ **Register Allocation :** Deciding names to be stored in registers.

⇒ **Register Assignment :** Deciding register to store values.

⇒ **Code Generation from DAG :**

It can be done by :

(*a*) Rearranging order

(*b*) Heuristic Ordering

(*c*) Labeling algorithm

⇒ **Peephole Optimization :** It improves the code locally. Short sequence of target code instructions are examined and are replaced by faster sequences.

REVIEW QUESTIONS

1. What do you know about object program ? Discuss it and also explain various problems in code generation ?
2. Explain Peephole optimization in Detail.
3. Write short note on Machine Model
4. Explain Register Allocation & Register Assignment in detail.