1. Explain code optimization technique.

1. **Common sub expressions elimination**
   - Compile time evaluation means shifting of computations from run time to compile time.
   - There are two methods used to obtain the compile time evaluation.

   Folding
   - In the folding technique the computation of constant is done at compile time instead of run time.

     Example : length = (22/7)*d
   - Here folding is implied by performing the computation of 22/7 at compile time.

   Constant propagation
   - In this technique the value of variable is replaced and computation of an expression is done at compilation time.

     Example:
     ```
         pi = 3.14; r = 5;
         Area = pi * r * r;
     ```
   - Here at the compilation time the value of pi is replaced by 3.14 and r by 5 then computation of 3.14 * 5 * 5 is done during compilation.
2. **Common sub expressions elimination**
   - The common sub expression is an expression appearing repeatedly in the program which is computed previously.
   - If the operands of this sub expression do not get changed at all then result of such sub expression is used instead of re-computing it each time.

     Example:
     ```
     t1 := 4 * i                 t1=4*i
     t2 := a[t1]                 t2=a[t1]
     t3 := 4 * j       ⟹         t3=4*j
     t4 : = 4 * i                t5=n
     t5:= n                      t6=b[t1]+t5
     t6 := b[t4]+t5
     ```
   - The common sub expression t4:=4*i is eliminated as its computation is already in t1 and value of i is not been changed from definition to use.
3. **Variable propagation**
   - Variable propagation means use of one variable instead of another.

     Example:

     x = pi;

     area = x * r * r;
   - The optimization using variable propagation can be done as follows, area = pi * r * r;
   - Here the variable x is eliminated. Here the necessary condition is that a variable must be assigned to another variable or some constant.
4. **Code movement**
   - There are two basic goals of code movement:

I. To reduce the size of the code.
II. To reduce the frequency of execution of code.

Example:

```
for(i=0;i<=10;i++)                 temp=y*5
{                                  for(i=0;i<=10;i++)
     x=y*5;         ==>            {
     k=(y*5)+50;                      x=temp;
}                                     k=(temp) + 50;
                                   }
```

Loop invariant computation

- Loop invariant optimization can be obtained by moving some amount of code outside the loop and placing it just before entering in the loop.
- This method is also called code motion.

Example:

```
While(i<=max-1)                        N=max-1;
{                                      While(i<=N)
     sum=sum+a[i];      ==>            {
}                                                sum=sum+a[i];
                                       }
```

5. **Strength reduction**
   - Strength of certain operators is higher than others.
   - For instance strength of * is higher than +.
   - In this technique the higher strength operators can be replaced by lower strength operators.
   - Example:

```
for(i=1;i<=50;i++)
{
          count = i*7;
}
```

   - Here we get the count values as 7, 14, 21 and so on up to less than 50.
   - This code can be replaced by using strength reduction as follows

```
temp=7;
for(i=1;i<=50;i++)
{
          count = temp;
          temp = temp+7;
}
```

6. **Dead code elimination**
   - A variable is said to be **live** in a program if the value contained into is subsequently.
   - On the other hand, the variable is said to be **dead** at a point in a program if the value contained into it is never been used. The code containing such a variable supposed to be a dead code. And an optimization can be performed by eliminating such a dead

code.

Example:

```
i=0;
if(i==1)
{
        a=x+5;
}
```

- If statement is a dead code as this condition will never get satisfied hence, statement can be eliminated and optimization can be done.

2. Explain Peephole optimization.

Definition: Peephole optimization is a simple and effective technique for locally improving target code. This technique is applied to improve the performance of the target program by examining the short sequence of target instructions (called the peephole) and replacing these instructions by shorter or faster sequence whenever possible. Peephole is a small, moving window on the target program.

Characteristics of Peephole Optimization

- The peephole optimization can be applied on the target code using following characteristic.

1. **Redundant instruction elimination.**
   - Especially the redundant loads and stores can be eliminated in this type of transformations.

     Example:

     ```
     MOV R0,x
     MOV x,R0
     ```
   - We can eliminate the second instruction since x is in already R0. But if MOV x, R0 is a label statement then we cannot remove it.
2. **Unreachable code.**
   - Especially the redundant loads and stores can be eliminated in this type of transformations.
   - An unlabeled instruction immediately following an unconditional jump may be removed.
   - This operation can be repeated to eliminate the sequence of instructions.

     Example:

     ```
     #define debug 0
     If(debug) {
             Print debugging information
     }
     ```

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

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

- One obvious peephole optimization is to eliminate jumps over jumps. Thus no matter what the value of debug, can be replaced by:

  If debug≠1 goto L2
  Print debugging information
  L2:
- Now, since debug is set to 0 at the beginning of the program, constant propagation should replace by

  If 0≠1 goto L2
  Print debugging information
  L2:
- As the argument of the first statement of 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.

3. **Flow of control optimization.**
   - 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: goto L2

     By the sequence

     Goto L2
     …….
     L1: goto L2
   - If there are 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
4. **Algebraic simplification.**
   - Peephole optimization is an effective technique for algebraic simplification.
   - The statements such as x = x + 0 or x := x* 1 can be eliminated by peephole optimization.
5. **Reduction in strength**
   - Certain machine instructions are cheaper than the other.
   - In order to improve performance of the intermediate code we can replace these instructions by equivalent cheaper instruction.
   - For example, x^2 is cheaper than x * x. Similarly addition and subtraction is cheaper than

multiplication and division. So we can add effectively equivalent addition and subtraction for multiplication and division.

6. **Machine idioms**
   - The target instructions have equivalent machine instructions for performing some operations.
   - Hence we can replace these target instructions by equivalent machine instructions in order to improve the efficiency.
   - Example: Some machines have auto-increment or auto-decrement addressing modes. These modes can be used in code for statement like i=i+1.

3. Loops in flow graphs.

1. **Dominators**
   - In a flow graph, a node d dominates n if every path to node n from initial node goes through d only. This can be denoted as'd dom n'.
   - Every initial node dominates all the remaining nodes in the flow graph. Similarly every node dominates itself.

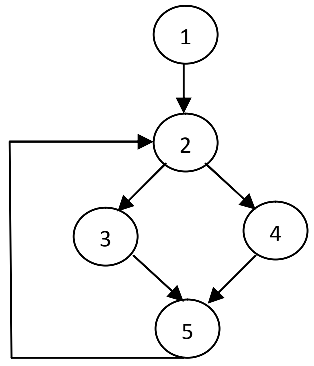


Fig.7.1. Dominators

   - Node 1 is initial node and it dominates every node as it is initial node.
   - Node 2 dominates 3, 4 and 5.
   - Node 3 dominates itself similarly node 4 dominates itself.

2. **Natural loops**
   - Loop in a flow graph can be denoted by n→d such that d dom n. This edge is called back edges and for a loop there can be more than one back edge. If there is p → q then q is a head and p is a tail. And head dominates tail.

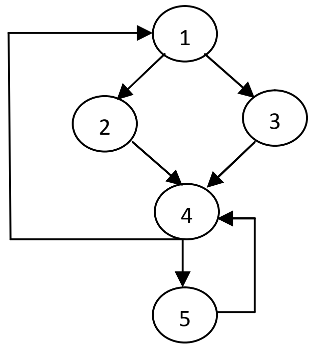


Fig.7.2. Natural loops

- The loop in above graph can be denoted by 4→1 i.e. 1 dom 4. Similarly 5→4 i.e. 4 dom 5.
- The natural loop can be defined by a back edge n→d such there exist a collection of all the node that can reach to n without going through d and at the same time d also can be added to this collection.

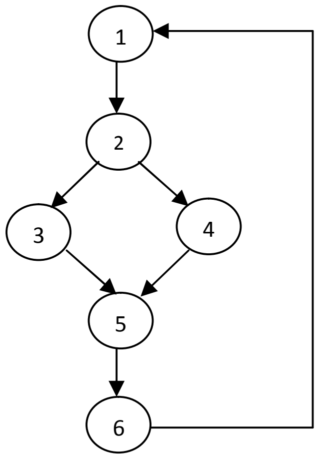


Fig.7.3. Natural loop

- 6→1 is a natural loop because we can reach to all the remaining nodes from 6.

3. **Inner loops**
   - The inner loop is a loop that contains no other loop.
   - Here the inner loop is 4→2 that mean edge given by 2-3-4.

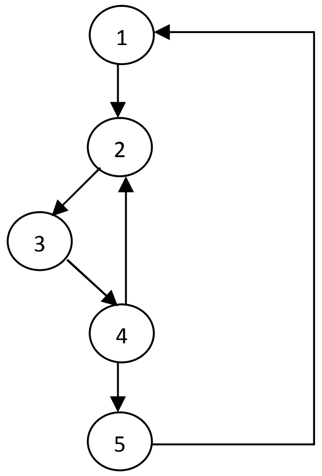


Fig.7.4. Inner loop

4. **Pre-header**
   - The pre-header is a new block created such that successor of this block is the block. All the computations that can be made before the header block can be made the pre-header block.

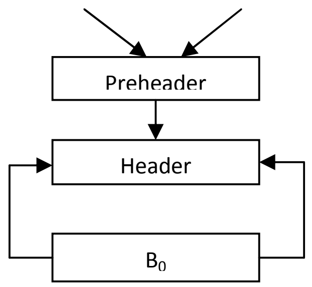


Fig.7.5. Preheader

5. **Reducible flow graph**
   - The reducible graph is a flow graph in which there are two types of edges forward edges and backward edges. These edges have following properties,
     I. The forward edge forms an acyclic graph.
     II. The back edges are such edges whose head dominates their tail.

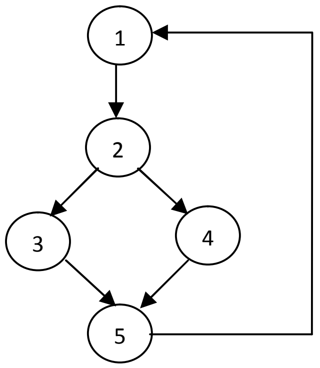


Fig.7.6. Reducible flow graph

Can be reduced as

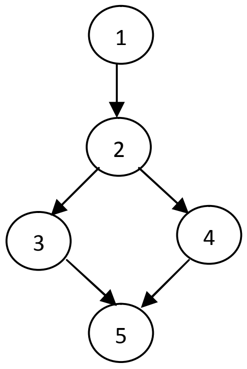


Fig.7.7. Reduced flow graph

   - The above flow graph is reducible. We can reduce this graph by removing the edge from 3 to 2. Similarly by removing the back edge from 5 to 1. We can reduce above flow graph and the resultant graph is a cyclic graph.

6. **Non-reducible flow graph**
   - A non reducible flow graph is a flow graph in which:
     I. There are no back edges.
     II. Forward edges may produce cycle in the graph.

Example: Following flow graph is non-reducible.

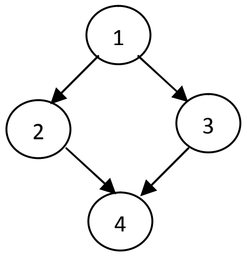


Fig.7.8. Non-Reducible flow graph

4. Global data flow analysis.

- Data flow equations are the equations representing the expressions that are appearing in the flow graph.
- Data flow information can be collected by setting up and solving systems of equations that relate information at various points in a program.
- The data flow equation written in a form of equation such that,

  out [S] = gen[S] U (in[S] -kill[S])
- And can be read as "the information at the end of a statement is either generated within a statement, or enters at the beginning and is not killed as control flows through the statement".
- The details of how dataflow equations are set up and solved depend on three factors.

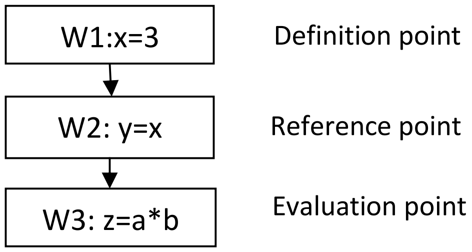
  I. The notions of generating and killing depend on the desired information, i.e., on the data flow analysis problem to be solved. Moreover, for some problems, instead of proceeding along with flow of control and defining out[s] in terms of in[s], we need to proceed backwards and define in[s] in terms of out[s].

  II. Since data flows along control paths, data-flow analysis is affected by the constructs in a program. In fact, when we write out[s] we implicitly assume that there is unique end point where control leaves the statement; in general, equations are set up at the level of basic blocks rather than statements, because blocks do have unique end points.

  III. There are subtleties that go along with such statements as procedure calls, assignments through pointer variables, and even assignments to array variables.

5. Data Flow Properties.

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



1. **Available expression**
    - An expression x+y is available at a program point w if and only if along all paths are reaching to w.
        I. The expression x+y is said to be available at its evaluation point.
        II. The expression x+y is said to be available if no definition of any operand of the expression (here either x or y) follows its last evaluation along the path. In other word, if neither of the two operands get modified before their use.

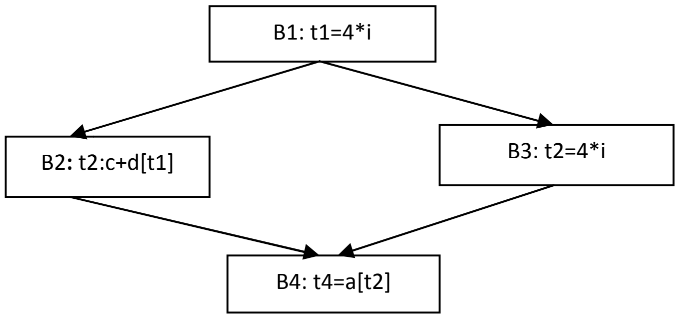


Fig.7.9. Available expression

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

2. **Reaching definition**
    - A definition D reaches at the point P if there is a path from D to P if there is a path from D to P along which D is not killed.
    - A definition D of variable x is killed when there is a redefinition of x.

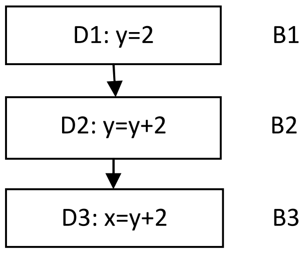


Fig.7.10. Reaching definition

- The definition D1 is reaching definition for block B2, but the definition D1 is not reaching definition for block B3, because it is killed by definition D2 in block B2.

3. **Live variable**
    - A live variable x is live at point p if there is a path from p to the exit, along which the value of x is used before it is redefined. Otherwise the variable is said to be dead at the point.
4. **Busy expression**
    - An expression *e* is said to be busy expression along some path pi..pj if and only if an evaluation of *e* exists along some path pi…pj and no definition of any operand exist before its evaluation along the path.

1. Role of code generator.

- The final phase of compilation process is code generation.
- It takes an intermediate representation of the source program as input and produces an equivalent target program as output.

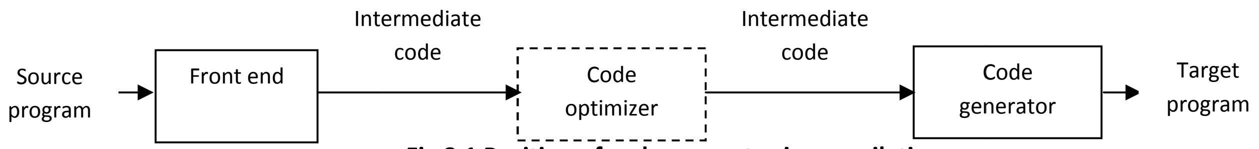


Fig 8.1 Position of code generator in compilation

- Target code should have following property,
    1. Correctness
    2. High quality
    3. Efficient use of resources of target code
    4. Quick code generation

2. Issues in the design of code generation.

Issues in design of code generator are:

1. **Input to the Code Generator**
    - Input to the code generator consists of the intermediate representation of the source program.
    - There are several types for the intermediate language, such as postfix notation, quadruples, and syntax trees or DAGs.
    - The detection of semantic error should be done before submitting the input to the code generator.
    - The code generation phase requires complete error free intermediate code as an input.
2. **Target program**
    - 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.
    - Producing a relocatable machine language program as output is that the subroutine can be compiled separately. A set of relocatable object modules can be linked together and loaded for execution by a linking loader.
    - 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.
3. **Memory management**
    - Mapping names in the source program to addresses of data objects in run time memory is done cooperatively by the front end and the code generator.
    - We assume that a name in a three-address statement refers to a symbol table entry for the name.

- From the symbol table information, a relative address can be determined for the name in a data area.

4. **Instruction selection**
   - If we do not care about the efficiency of the target program, instruction selection is straightforward. It requires special handling. For example, the sequence of statements
     a := b + c
     d := a + e
     would be translated into

```
MOV   b, R0
ADD   c, R0
MOV   R0, a
MOV   a, R0
ADD   e, R0
MOV   R0, d
```

   - Here the fourth statement is redundant, so we can eliminate that statement.
5. **Register allocation**
   - If the instruction contains register operands then such a use becomes shorter and faster than that of using in memory.
   - The use of registers is often subdivided into two sub problems:
   - During register allocation, we select the set of variables that will reside in registers at a point in the program.
   - 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 value.
   - Mathematically the problem is NP-complete.
6. **Choice of evaluation**
   - 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.
7. **Approaches to code generation**
   - The most important criterion for a code generator is that it produces 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.

3. The target machine and instruction cost.

- Familiarity with the target machine and its instruction set is a prerequisite for designing a good code generator.
- We will assume our target computer models a three-address machine with load and store operations, computation operations, jump operations, and conditional jumps. The

underlying computer is a byte-addressable machine with n general-purpose registers, R_0, R_1, . . . , R_n

- The two address instruction of the form *op source, destination*
- It has following opcodes,

  MOV (move source to destination)
  ADD (add source to destination)
  SUB (subtract source to destination)
- The address modes together with the assembly language forms and associated cost as follows:

Mode	Form	Address	Extra cost
Absolute	M	M	1
Register	R	R	0
Indexed	k(R)	k +contents(R)	1
Indirect register	*R	contents(R)	0
Indirect indexed	*k(R)	contents(k + contents(R))	1

Table 8.1 Addressing modes

Instruction cost:

- The instruction cost can be computed as one plus cost associated with the source and destination addressing modes given by "extra cost".
- Calculate cost for following:

```
MOV B,R0
ADD C,R0
MOV R0,A
```

Instruction cost,

```
MOV B,R0→cost = 1+1+0=2
ADD C,R0→cost = 1+1+0=2
MOV R0,A→cost = 1+0+1=2
________________________________
Total cost=6
```

4. 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
```

- Some terminology used in basic blocks are given below:
- 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 be *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: 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**, for that we use the following rules:*
   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: Program to compute dot product

```
begin
        prod := 0;
i := 1;
do
                prod := prod + a[t1] * b[t2];
                i := i+1;
while  i<= 20
end
```

Three address code for the above program,

```
(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)
```

- Let us apply an algorithm to the three-address code 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.

- Therefore, statements (1) and (2) form a basic block.
- The remainder of the program beginning with statement (3) forms a second basic block.

5. Transformations on basic block

- 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 the code.
- There are two important classes of local transformations that can be applied to basic block. These are,
    1. Structure preserving transformation.
    2. Algebraic transformation.

1. **Structure Preserving Transformations**

    The primary structure-preserving transformations on basic blocks are,

    A. **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, 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.

    B. **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.

    C. **Renaming of 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.
    - In fact, we can always transform a basic block into an equivalent block in which each statement that defines a temporary defines a new temporary. We call such a basic block a *normal-form* block.

    D. **Interchange of two independent adjacent 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.

2. **Algebraic transformation**
   - Countless algebraic transformation can be used to change the set of expressions computed by the basic block into an algebraically equivalent set.
   - The useful ones are those that simplify expressions or replace expensive operations by cheaper one.
   - Example: x=x+0 or x=x+1 can be eliminated.
3. **Flow graph**
   - A graph representation of three-address statements, called a **flow graph**, is useful for understanding code-generation algorithms.
   - Nodes in the flow graph represent computations, and the edges represent the flow of control.
   - Example of flow graph for following 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<=20 goto (3)
```

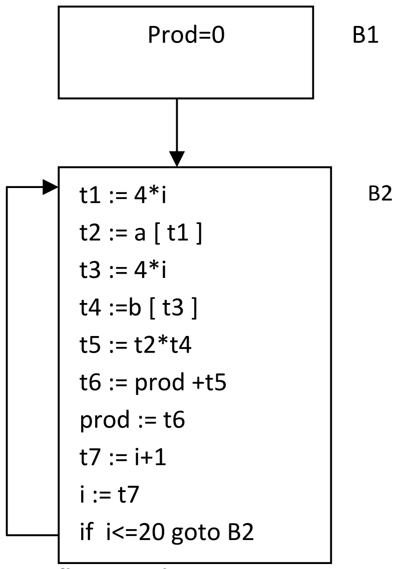


Fig 8.2 flow graph

6. Next-Use information.

- The next-use information is a collection of all the names that are useful for next subsequent statement in a block. The use of a name is defined as follows,
- Consider a statement,

  x := i
  j := x op y
- That means the statement j uses value of x.
- The next-use information can be collected by making the backward scan of the programming code in that specific block.

Storage for Temporary Names

- For the distinct names each time a temporary is needed. And each time a space gets allocated for each temporary. To have optimization in the process of code generation we pack two temporaries into the same location if they are not live simultaneously.
- Consider three address code as,

```
t1   := a * a            t1  := a * a
t2   := a * b            t2  := a * b
t3   := 4 * t2    ⟹      t2  := 4 * t2
t4   := t1+t3            t1  := t1+t2
t5   := b * b            t2  := b * b
t6   := t4+t5            t1  := t1+t2
```

7. Register and address descriptors.

- The code generator algorithm uses descriptors to keep track of register contents and addresses for names.
- **Address descriptor** stores the location where the current value of the name can be found at run time. The information about locations can be stored in the symbol table and is used to access the variables.
- **Register descriptor** is used to keep track of what is currently in each register. The register descriptor shows that initially all the registers are empty. As the generation for the block progresses the registers will hold the values of computation.

8. Register allocation and assignment.

- Efficient utilization of registers is important in generating good code.
- There are four strategies for deciding what values in a program should reside in a registers and which register each value should reside. Strategies are,

1. **Global register allocation**
   - Following are the strategies adopted while doing the global register allocation.
   - The global register allocation has a strategy of storing the most frequently used variables in fixed registers throughout the loop.
   - Another strategy is to assign some fixed number of global registers to hold the most active values in each inner loop.
   - The registers are not already allocated may be used to hold values local to one block.
   - In certain languages like C or Bliss programmer can do the register allocation by using register declaration to keep certain values in register for the duration of the procedure.
2. **Usage count**
   - The usage count is the count for the use of some variable x in some register used in any basic block.

- The usage count gives the idea 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 the Loop L in some basic block B can be given as,

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

- Where use(x,B) is number of times x used in block B prior to any definition of x
- live(x,B) =1 if x is live on exit from B; otherwise live(x)=0.

3. **Register assignment for outer loop**
   - Consider that there are two loops L1 is outer loop and L2 is an inner loop, and allocation of variable a is to be done to some register. The approximate scenario is as given below,

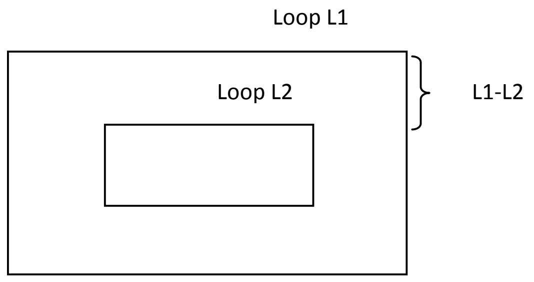


Fig 8.3 Loop represenation

Following criteria should be adopted for register assignment for outer loop,

- If a is allocated in loop L2 then it should not be allocated in L1 - L2.
- If a is allocated in L1 and it is not allocated in L2 then store a on entrance to L2 and load a while leaving L2.
- If a is allocated in L2 and not in L1 then load a on entrance of L2 and store a on exit from L2.

4. **Register allocation for graph coloring**

The graph coloring works in two passes. The working is as given below,

- In the first pass the specific machine instruction is selected for register allocation. For each variable a symbolic register is allocated.
- In the second pass the register inference graph is prepared. In register inference graph each node is a symbolic registers and an edge connects two nodes where one is live at a point where other is defined.
- Then a graph coloring technique is applied for this register inference graph using k-color. The k-colors can be assumed to be number of assignable registers. In graph coloring technique no two adjacent nodes can have same color. Hence in register inference graph using such graph coloring principle each node (actually a variable) is assigned the symbolic registers so that no two symbolic registers can interfere with each other with assigned physical registers.

9. DAG representation of basic blocks.

- The directed acyclic graph is used to apply transformations on the basic block.

- A DAG gives a picture of how the value computed by each statement in a basic block used in a subsequent statements of the block.
- To apply the transformations on basic block a DAG is constructed from three address statement.
- A DAG can be constructed for the following type of labels on nodes,
    1. Leaf nodes are labeled by identifiers or variable names or constants. Generally leaves represent r-values.
    2. Interior nodes store operator values.
    3. Nodes are also optionally given a sequence of identifiers for label.
- The DAG and flow graphs are two different pictorial representations. Each node of the flow graph can be represented by DAG because each node of the flow graph is a basic block.

  Example:

  sum = 0;

  for (i=0;i< = 10;i++)

  sum = sum+a[t1];

  Solution :

  The three address code for above code is

1. sum :=0
2. i:=0
3. t1 := 4*i
4. t2:= a[t1]
5. t3 := sum+t2
6. sum := t3
7. t4 := i+1;
8. i:= t4
9. if i<=10 goto (3)

B1

B2

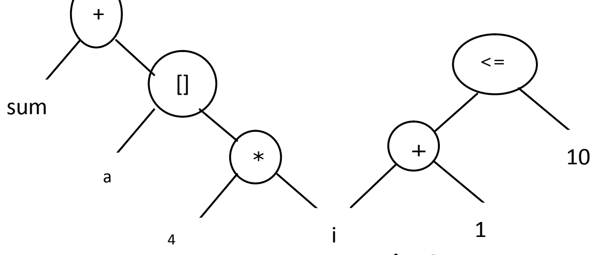


Fig 8.4 DAG for block B2

Algorithm for Construction of DAG

- We assume the three address statement could of following types,

  Case (i) x:=y op z

  Case (ii)x:=op y

  Case (iii) x:=y

- With the help of following steps the DAG can be constructed.
  Step 1: If y is undefined then create node(y). Similarly if z is undefined create a node(z)
  Step 2: For the case(i) create a node(op) whose left child is node(y) and node(z) will be the right child. Also check for any common sub expressions. For the case(ii) determine whether is a node labeled op,such node will have a child node(y). In case(iii) node n win be node(y).
  Step 3: Delete x from list of identifiers for node(x). Append x to the list of attached identifiers for node n found in 2.

Applications of DAG

The DAGs are used in,

1. Determining the common sub-expressions.
2. Determining which names are used inside the block and computed outside the block.
3. Determining which statements of the block could have their computed value outside the block.
4. Simplifying the list of quadruples by eliminating the common sub-expressions and not performing the assignment of the form x:=y unless and until it is a must.

10. Generating code from DAGs.

- Methods generating code from DAG as shown in Figure 8.5.

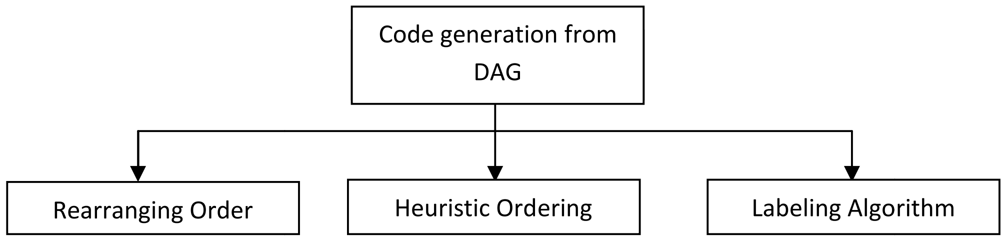


Fig. 8.5. Methods to generate code from DAG

1. Rearranging Order

- The order of three address code affects the cost of the object code being generated. In the senses that by changing the order in which computations are done we can obtain the object code with minimum cost.
- Consider the following code,

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

- The code can be generated by translating the three address code line by line.

```
MOV a, R0
ADD b, R0
MOV c, R1
ADD d, R1
MOV R0, t1
MOV e, R0
```

```
SUB R1, R0
MOV t1, R1
SUB R0, R1
MOV R1, t4
```

- Now if we change the sequence of the above three address code.

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

- Then we can get improved code as

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

2. Heuristic ordering

- The heuristic ordering algorithm is as follows:

```
1.  Obtain all the interior nodes. Consider these interior nodes as unlisted nodes.
2.  while( unlisted interior nodes remain)
3.          {
4.          pick up an unlisted node n, whose parents have been listed
5.          list n;
6.          while(the leftmost child m of n has no unlisted parent AND is not leaf
7.                  {
8.                  List m;
9.                  n=m;
10.                 }
11.         }
```

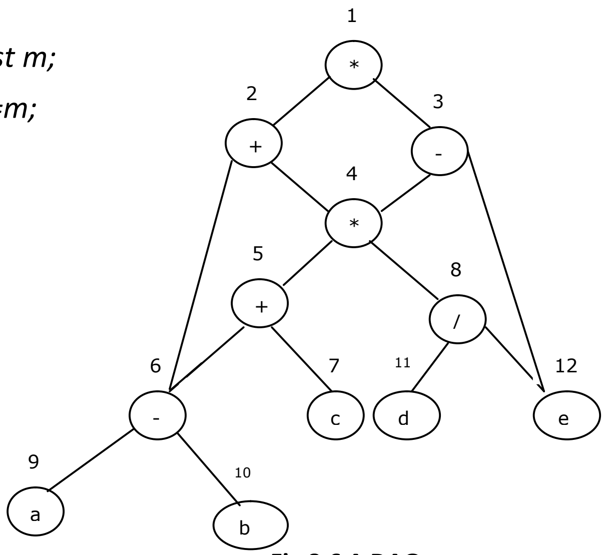


Fig 8.6 A DAG

- The DAG is first numbered from top to bottom and from left to right. Then consider the unlisted interior nodes 1 2 3 4 5 6 8.
- Initially the only node with unlisted parent is 1.(Set n=1 by line 4 of algorithm)
- Now left argument of 1 is 2 and parent of 2 is 1 which is listed. Hence list 2. Set n=2 by line 7) of algorithm

  $\underline{1\ 2}\ 3\ 4\ 5\ 6\ 8$
- Now we will find the leftmost node of 2 and that is 6. But 6 has unlisted parent 5. Hence we cannot select 6.
- We therefore can switch to 3. The parent of 3 is 1 which is listed one. Hence list 3 set n=3

  $\underline{1\ 2\ 3}\ 4\ 5\ 6\ 8$
- The left of 3 is 4. As parent of 4 is 3 and that is listed hence list 4. Left of 4 is 5 which has listed parent (i.e. 4) hence list 5. Similarly list 6

  $\underline{1\ 2\ 3\ 4\ 5\ 6}\ 8$
- As now only 8 is remaining from the unlisted interior nodes we will list it.
- Hence the resulting list is 1 2 3 4 5 6 8.
- Then the order of computation is decided by reversing this list.
- We get the order of evaluation as 8 6 5 4 3 2 1.
- That also means that we have to perform the computations at these nodes in the given order,

```
T8=d/e
T6=a-b
T5=t6+c
T4=t5*t8
T3=t4-e
T2=t6+t4
T1=t2*t3
```

3. Labeling algorithm

- The labeling algorithm generates the optimal code for given expression in which minimum registers are required.
- Using labeling algorithm the labeling can be done to tree by visiting nodes in bottom up order.
- By this all the child nodes will be labeled its parent nodes.
- For computing the label at node n with the label L1 to left child and label L2 to right child as,

  Label (n) = max(L1,L2) if L1 not equal to L2

  Label(n) = L1+1 if L1=L2
- We start in bottom-up fashion and label left leaf as 1 and right leaf as 0.

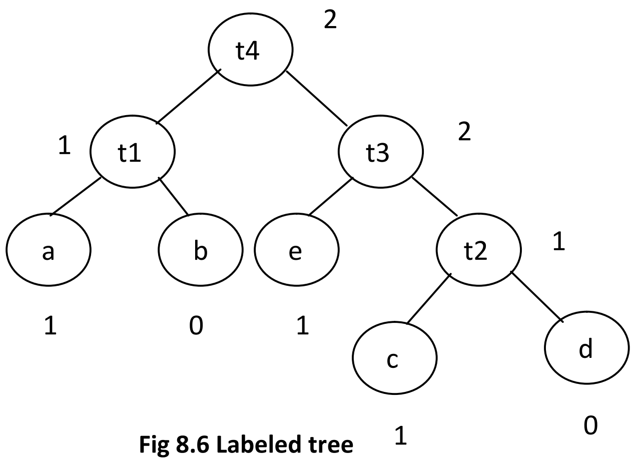


Fig 8.6 Labeled tree