UNIT – V

Intermediate code generation

Objective

To understand various intermediate code forms and machine independent optimization techniques.

Syllabus

Intermediate code- Three address code, quadruples, triples, abstract syntax trees.

Machine independent code optimization- Common sub expression elimination, constant folding, copy propagation, dead code elimination, strength reduction, loop optimization, procedure inlining, basic blocks, DAG Construction and its applications, Control Flow Graph.

Learning Outcomes

Students will be able to

- Understand intermediate code forms such as polish notation,syntax trees and three-address code .
- Write three address code for various language constructs.
- Construct basic blocks for the given language constructs.
- Construct Flow graph for the given basic block.
- Construct DAG for the given basic block and expression.

Learning Material

5.1 Intermediate Code

- In the analysis-synthesis model of a compiler, the front end analyzes a source program and creates an intermediate representation, from which the back end generates target code.
- Ideally, details of the source language are confined to the front end, and details of the target machine to the back end. The source code is translated into a language, called **Intermediate code or intermediate *text*** which is intermediate in complexity between a high level programming language and a machine code.

  Benefits of using a machine independent intermediate form are:
- Retargeting is facilitated, a compiler for different machine can be created by attaching a back end for the new machine to an existing front end.
- A machine-independent code optimizer can be applied to the intermediate representation.

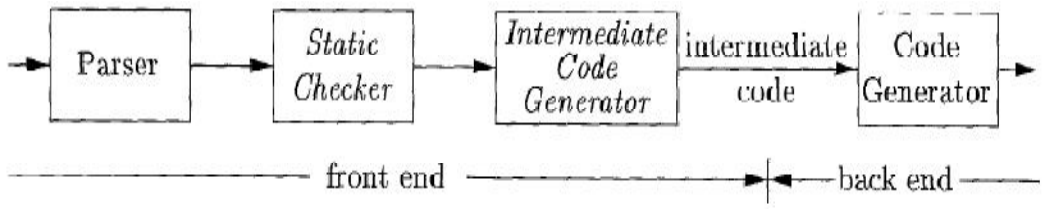


Fig1: Position of Intermediate Code Generator

5.1.1 Intermediate code representations

- Postfix (or) reverse polish notation
- Syntax tree or parse tree
- Three address code

1)Postfix (or) reverse polish notation:

- The postfix notation for the expression places the operator at the right end.
- If e1 and e2 are any postfix expressions, and □ is any binary operator, the result of applying □ to the values denoted by e1 and e2 is indicated in postfix notation by e1e2□.

Example: (a+b)*c in postfix notation is ab+c*.

2)Syntax tree or Parse tree:

- A syntax tree depicts the natural hierarchical structure of a source program.
- A syntax tree for the assignment statement a:= b*-c + b*-c is

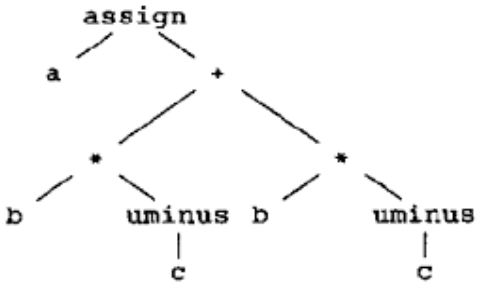


Fig2: Parse Tree

3)Three Address Code:

- Three address code is a sequence of statements of the general form **x := y op z**
- where x ,y and z are names, constants or compiler-generated temporaries;
- **op** stands for any operator.
- In three-address code, there is at most one operator on the right side of an instruction; that is, no built-up arithmetic expressions are permitted.
- A source language expression like x+y*z might be translated into the sequence of three-address instructions
    - t1=y*z
    - t2=x+t1
- where **tl** and **t2** are compiler-generated temporary names.

Symbolic labels will be used by instructions that alter the flow of control. A symbolic label represents the index of a three-address instruction in the sequence of instructions. Actual indexes can be substituted for the labels, either by making a separate pass or by "back patching".

5.2 Three address Code

Three address code is represented as record structure with fields for operator and operands. These records can be stored as array or linked list.

Most common implementations of three address code are-

- **Quadruples**
- **Triples**
- **Indirect triples**
- **Abstract Syntax Tree**

5.2.1 QUADRUPLES

Quadruples consists of four fields in the record structure. One field to store operator op, two fields to store operands or arguments arg1and arg2 and one field to store result res.

res = arg1 op arg2

Example: a = b + c

b is represented as arg1, c is represented as arg2, + as op and a as res.

Unary operators like '-'do not use agr2. Operators like param do not use agr2 nor result. For conditional and unconditional statements res is label. Arg1, arg2 and res are pointers to symbol table or literal table for the names.

Example: a = -b * d + c + (-b) * d

Three address code for the above statement is as follows

t1 = - b

t2 = t1 * d

t3 = t2 + c

t4 = - b

t5 = t4 * d

t6 = t3 + t5

a = t6

Quadruples for the above example is as follows

Op	Arg1	Arg2	Result
-	b		t1
*	t1	d	t2
+	t2	c	t3
-	b		t4
*	t4	d	t5

+	t3	t5	t6
=	t6		a

Table 1: Quadruples table for Three address code

5.2.2 TRIPLES

Triples uses only three fields in the record structure. One field for operator, two fields for operands named as arg1 and arg2. Value of temporary variable can be accessed by the position of the statement the computes it and not by location as in quadruples.

Example: a = -b * d + c + (-b) * d

Triples for the above example is as follows

Stmt no	Op	Arg1	Arg2
(0)	-	b	
(1)	*	d	(0)
(2)	+	c	(1)
(3)	-	b	
(4)	*	d	(3)
(5)	+	(2)	(4)
(6)	=	a	(5)

Table 2: Triple Table

Arg1 and arg2 may be pointers to symbol table for program variables or literal table for constant or pointers into triple structure for intermediate results.

Example: Triples for statement x[i] = y which generates two records is as follows

Stmt no	Op	Arg1	Arg2
(0)	[ ]=	x	i
(1)	=	(0)	y

Table 3: Triples for statement x[i] = y which generates two records

Triples for statement x = y[i] which generates two records is as follows

Stmt no	Op	Arg1	Arg2
(0)	=[ ]	y	i
(1)	=	x	(0)

Table 4:Triples for statement x = y[i] which generates two records

Triples are alternative ways for representing syntax tree or Directed acyclic graph for program defined names.

5.2.3 Indirect Triples

Indirect triples are used to achieve indirection in listing of pointers. That is, it uses pointers to triples than listing of triples themselves.

Example: a = -b * d + c + (-b) * d

Stmt no	
(0)	(10)
(1)	(11)
(2)	(12)
(3)	(13)
(4)	(14)
(5)	(15)
(6)	(16)

	Op	Arg1	Arg2
(10)	-	b	
(11)	*	d	(0)
(12)	+	c	(1)
(13)	-	b	
(14)	*	d	(3)
(15)	+	(2)	(4)
(16)	=	a	(5)

Table 5: Indirect triples

Conditional operator and operands. Representations include quadruples, triples and indirect triples.

Note: The conversion of all labels in three address statements to addresses of instructions is known as backpatching.

5.2.4 abstract syntax tree (AST):

An **abstract syntax tree** (**AST**) is a way of representing the **syntax** of a programming language as a hierarchical **tree**-like structure. This structure is used for

generating symbol tables for compilers and later code generation. The **tree** represents all of the constructs in the language and their subsequent rules.

5.3 BASIC BLOCKS

5.3.1Basic 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 any 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
```

5.3.2 Loop Optimizations:

- We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.
- Three techniques are important for loop optimization:
  - code motion, which moves code outside a loop;
  - Induction-variable elimination, which we apply to replace variables from inner loop.
  - Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition.

5.3.3 Various steps to Loop Optimization:

1) Convert the program into Three-address code
2) Break the code into basic blocks
3) Construct flow graph
4) Apply optimization
   a) Code Motion

b) Induction variable elimination

c) Strength reduction

Example:

Consider the following source code for dot product of two vectors a and b of length 20

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

Step1: Convert the program into Three-address code

The three-address code for the above source program is given as :

(1) prod := 0

(2) i := 1

(3) t1 := 4* i

(4) t2 := a[t1] /*compute a[i] */

(5) t3 := 4* i

(6) t4 := b[t3] /*compute b[i] */

(7) t5 := t2*t4

(8) t6 := prod+t5

(9) prod := t6

(10) t7 := i+1

(11) i := t7

(12) if i<=20 goto (3)

Step2: Break the code into basic blocks

5.3.4 Basic Block Construction

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, the first statements of basic blocks. The rules we use are of the following:

   - The first statement is a leader.
   - Any statement that is the target of a conditional or unconditional goto is a
   - leader.
   - Any statement that immediately follows a goto or conditional goto statement is a leader.

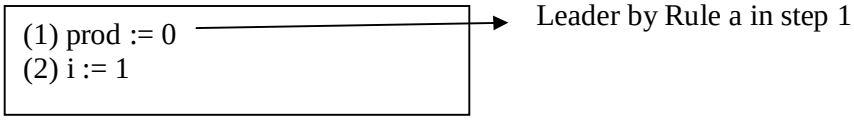
2. For each leader, its basic block consists of the leader and all statements up to but not including the next leader or the end of the program.

3. Any statement not placed in a block can never be executed and may now be removed if desired.

Basic block 1: Statement (1) to (2)

Basic block 2: Statement (3) to (11)

Basic block B1



Basic block B2

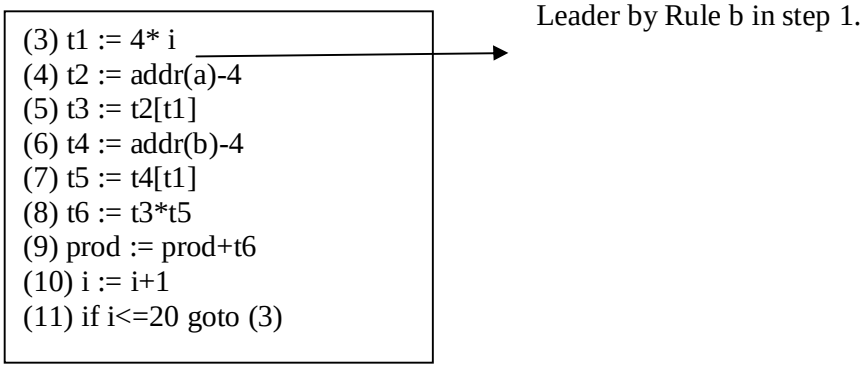


Fig: Basic block

Step3: Construct flow graph

5.3.5 Flow Graphs

- Flow graph is a directed graph containing the flow-of-control information for the set of basic blocks making up a program.
- The nodes of the flow graph are basic blocks. It has a distinguished initial node.
- E.g.: Flow graph for the vector dot product is given as follows:

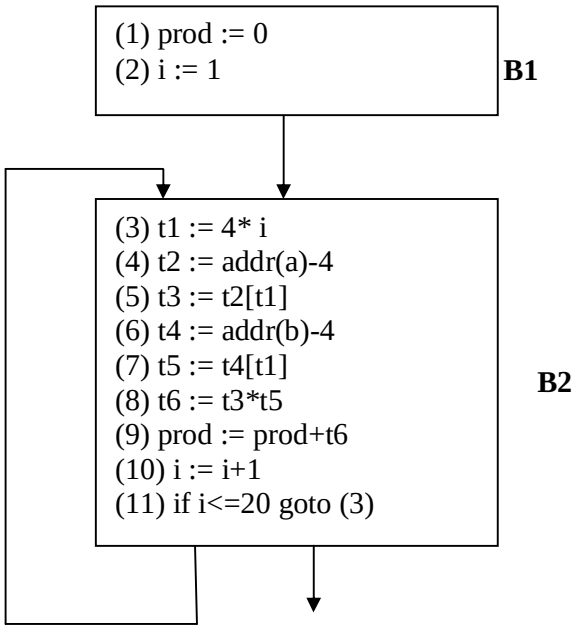


Fig 3:Flow graph for the vector dot product

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

Step 4: Apply optimization

5.4 Directed Acyclic Graph(DAG)

5.4.1 Variants of Syntax Trees:

- DAG A directed acyclic graph (DAG) for an expression identifies the common sub expressions (sub expressions that occur more than once) of the expression.
- DAG's can be constructed by using the same techniques that construct syntax trees.

- A DAG has leaves corresponding to atomic operands and interior nodes corresponding to operators.
- A node N in a DAG has more than one parent if N represents a common sub expression, so a DAG represents expressions concisely. It gives clues to compiler about the generating efficient code to evaluate expressions.

Example : DAG for the expression a + a * (b - c) + (b - c) * d is shown below.

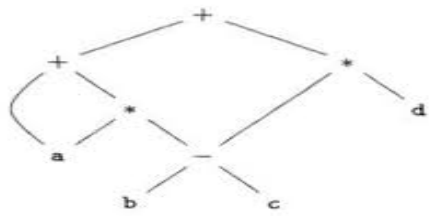


Fig 7: DAG for the expression a + a * (b - c) + (b - c) * d

5.4.2 DAG representation of Basic Blocks:

The first step in eliminating local common sub expressions is to detect the common sub expression in a basic block. The common sub expressions in a basic block can be automatically detected if we construct a directed acyclic graph (DAG). A DAG for a basic block is a directed acyclic graph with the following labels on nodes:

1. Leaves are labeled by unique identifiers, either variable names or constants.
2. Interior nodes are labeled by an operator symbol.
3. Nodes are also optionally given a sequence of identifiers for labels. The intention is that interior nodes represent computed values and the identifiers labeling a node are deemed to have that value.

5.4.3 DAG Construction

For constructing a basic block DAG, we make use of the function node(id), which returns the most recently created node associated with id. For every three-address statement $x = y\ op\ z$, $x = op\ y$, or $x = y$ in the block we:

do

{

1. If node(y) is undefined, create a leaf labeled y, and let node(y) be this node. If node(z) is undefined, create a leaf labeled z, and let that leaf be node(z). If the statement is of the form $x = op\ y$ or $x = y$, then if node(y) is undefined, create a leaf labeled y, and let node(y) be this node.
2. If a node exists that is labeled *op* whose left child is node(y) and whose right child is node(z) (to catch the common sub expressions), then return this node. Otherwise, create such a node, and return it. If the statement is of the form $x = op\ y$, then check if a node exists that is labeled *op* whose only child is node(y). Return this node. Otherwise, create such a node and return. Let the returned node be n.
3. Append x to the list of identifiers for the node n returned in step 2. Delete x from the list of attached identifiers for node(x), and set node(x) to be node n.

}

Therefore, we first go for a DAG representation of the basic block. And if the interior nodes in the DAG have more than one label, then those nodes of the DAG represent the common sub expressions in the basic block. After detecting these common sub expressions, we eliminate them from the basic block. The following example shows the elimination of local common sub expressions, and the DAG is shown in Figure 1.

Example 1:

1. $S1$: = 4 * I
2. $S2$: addr(A) − 4
3. $S3$: $S2$ [$S1$]
4. $S4$: 4 * I
5. $S5$: = addr(B) − 4
6. $S6$: = $S5$ [$S4$]
7. $S7$: = $S3$ * $S6$
8. $S8$: PROD + $S7$
9. PROD : = $S8$
10. $S9$: = I + 1

11. $I = S9$
12. if $I \leq 20$ goto (1).

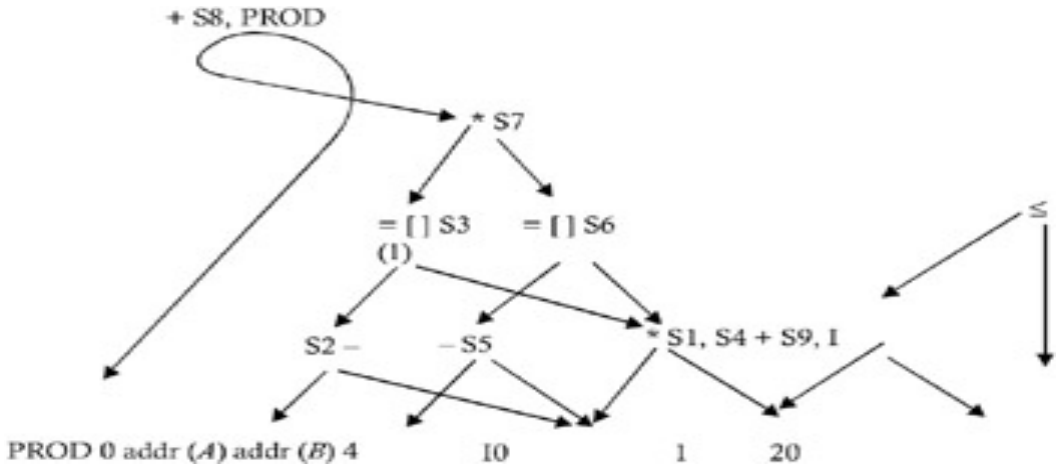


Fig 8: DAG representation of a basic block.

In fig, PROD 0 indicates the initial value of PROD, and $I0$ indicates the initial value of I. We see that the same value is assigned to $S8$ and PROD. Similarly, the value assigned to $S9$ is the same as I. And the value computed for $S1$ and $S4$ are the same; hence, we can eliminate these common subexpressions by selecting one of the attached identifiers (one that is needed outside the block). We assume that none of the temporaries is needed outside the block. The rewritten block will be:

1. $S1 := 4 * I$
2. $S2 := \text{addr}(A) - 4$
3. $S3 := S2\ [S1]$
4. $S5 := \text{addr}(B) - 4$
5. $S6 := S5\ [S1]$
6. $S7 := S3 * S6$
7. $\text{PROD} := \text{PROD} + S7$
8. $I := I + 1$
9. if $I \leq 20$ goto (1)

Example 2:

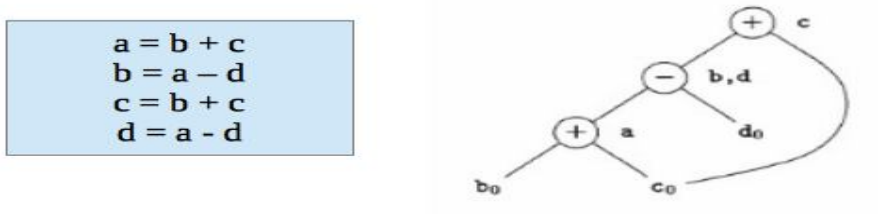


Fig9: DAG

5.4.4 Applications of DAG:

1. It detect common subexpressions.
2. We can determine which identifiers have their values used in the block.
3. We can determine which statements compute values which could be used outside the block

5.5 Machine independent code optimization

The code produced by the straight forward compiling algorithms can often be made to run faster or take less space, or both. This improvement is achieved by program transformations that are traditionally called optimizations. Compilers that apply code-improving transformations are called optimizing compilers.

Optimizations are classified into two categories. They are

- Machine independent optimizations
- Machine dependant optimizations

5.5.1 Machine independent optimizations:

- Machine independent optimizations are program transformations that improve the target code without taking into consideration any properties of the target machine.

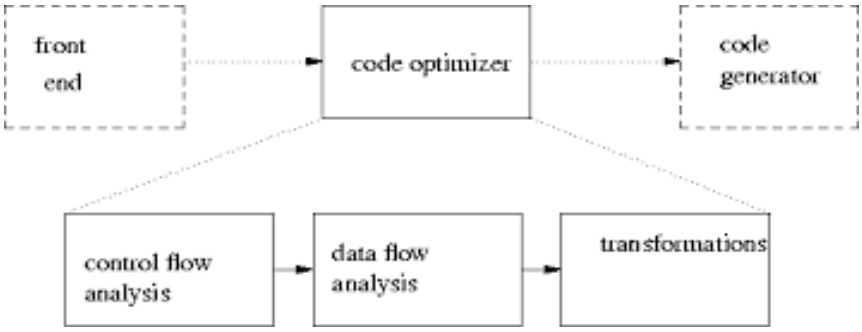


Fig 10: Organization for an Optimizing Compiler

5.5.2 Common Sub expressions elimination:

An occurrence of an expression E is called a common sub-expression if E was previously computed, and the values of variables in E have not changed since the previous computation. We can avoid re computing the expression if we can use the previously computed value.

Example 1:

t1: = 4*i

t2: = a [t1]

t3: = 4*j

t4: = 4*i

t5: = n

t6: = b [t4] +t5

The above code can be optimized using the common sub-expression elimination as

t1: = 4*i

t2: = a [t1]

t3: = 4*j

t5: = n

t6: = b [t1] +t5

Example 2: (Global common sub expression elimination)

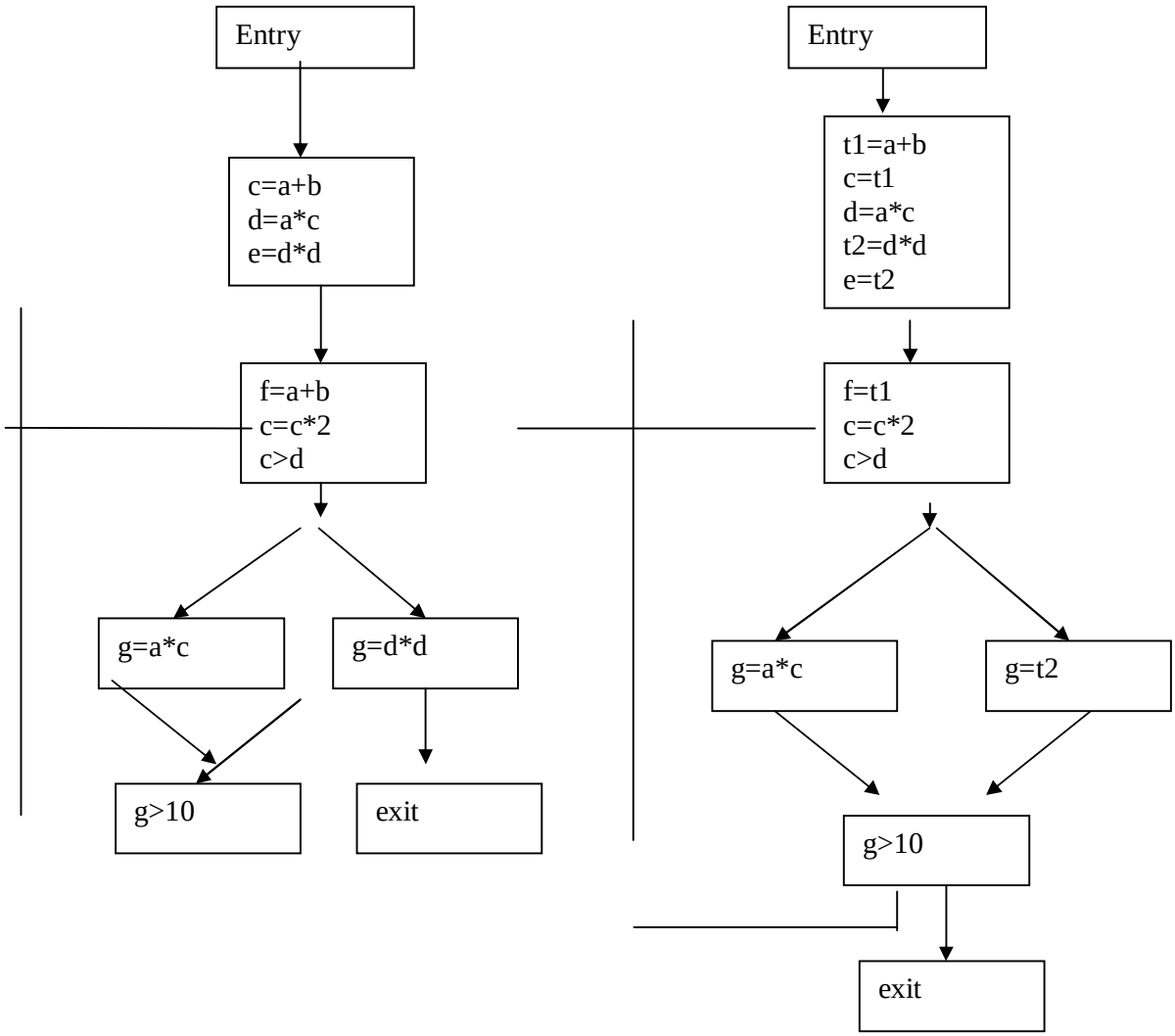


Fig6:Global common sub expression elimination

Here a+b and d*d are common sub expressions, throughout the flow graph.

a+b and d*d are called as available expressions.

5.5.3 Constant folding:

- Folding is the replacement of expressions that can be evaluated at compile time by their computed values. That is, if the values of all the operands of expression are

known to the compiler at compile time, the expression can be replaced by its computed value.

- One advantage of copy propagation is that it often turns the copy statement into dead
- code.
- Constant folding is also known as constant prorogation.
- Folding is a local optimization technique; folding operation can take place involving a variable that is common to a set of contiguous block in a program.

Example 1:

The statement A=2+3+A+C

can be replaced by A=5+A+C, where the value"5" has replaced the expression "2+3"

Example 2:

Pi=3.14157

a=pi/2

Can be rewritten as

Pi=3.14157

a=3.14157/2 can be replaced by

Pi= 3.14157

a=1.570 thereby eliminating a division operation.

5.5.4 Copy Propagation:

- Assignments of the form f : = g called copy statements, or copies for short.
- The idea behind the copy-propagation transformation is to use g for f, whenever possible after the copy statement f: = g.
- Copy propagation means use of one variable instead of another. This may not appear to be an improvement, but as we shall see it gives us an opportunity to eliminate x.

For example:

In block B

```
x := t3
a[t2] := t5
a[t4] := x
goto B2
```

The assignment x:=t3 in block B is a copy.

Copy propagation applied to B yields:

```
x := t3
a[t2] := t5
a[t4] := t3
goto B2
```

One **advantage** of copy propagation is that it often turns the copy statement into dead code. For example, copy propagation followed by dead-code elimination removes the assignment to x and transforms into:

```
a[t2] := t5
a[t4] := t3
goto B2
```

5.5.5 Dead-Code Elimination:

- A variable is live at a point in a program if its value can be used subsequently; otherwise, it is dead at that point.
- A related idea is dead or useless code, statements that compute values that never get used. While the programmer is unlikely to introduce any dead code intentionally, it may appear as the result of previous transformations.
- An optimization can be done by eliminating dead code.

Example 1:

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

Here, 'if' statement is dead code because this condition will never get satisfied.

Example 2:

The use of debug that is set to true or false at various points in the program and used in statements like

```
debug := false
if(debug)
print ……..
```

- By a data-flow analysis, it may be possible to deduce that each time the program reaches this statement, the value of debug is false.

If copy propagation replaces debug by false, then the print statement is dead because it cannot be reached. We eliminate both the test and printing from the object code

5.6 Loop Optimization

We now give a brief introduction to a very important place for optimizations, namely loops, especially the inner loops where programs tend to spend the bulk of their time. The running time of a program may be improved if we decrease the number of instructions in an inner loop, even if we increase the amount of code outside that loop.

Three techniques are important for loop optimization:

i) Code motion, which moves code outside a loop;

ii) Reduction in strength, which replaces and expensive operation by a cheaper one, such as a multiplication by an addition

5.6.1 Reduction In Strength:

- Reduction in strength replaces expensive operations by equivalent cheaper ones on the target machine. Certain machine instructions are considerably cheaper than others and can often be used as special cases of more expensive operators.
- In the example, the statement t1=4*i is costly when compared to t1=t1+4.
- So we are replacing t1=4*i with less costly statement t1=t1+4.

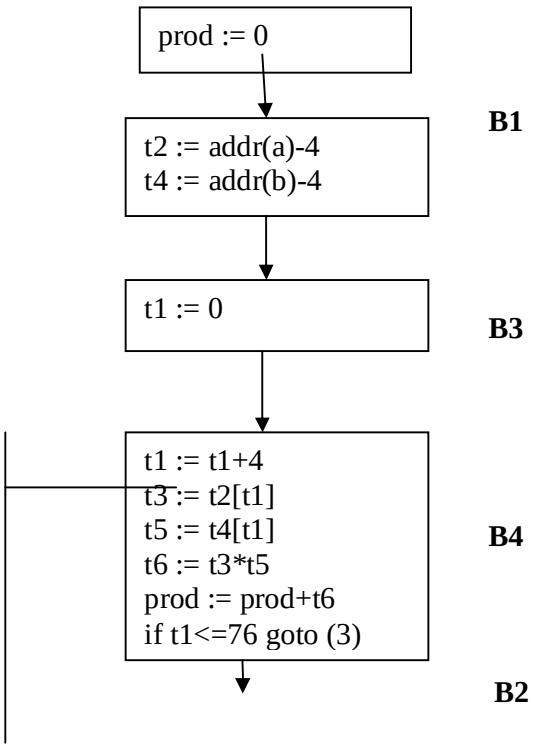


Fig 5:Flow graph for Reduction In Strength

- The values of i and t1 remain in lock-step i.e., at the assignment t1=4*i, i takes on the values 1,2…..20. Thus t1 takes the values 4,8…..80 immediately after each assignment to t. So we call such identifiers as *induction variables.*
- The assignment t1=4*i, replace the statement by t1=t1+4 and assign t1 :=0. This replacement will speed up the object code as addition takes less time than multiplication in many machines.
- We replace i<=20 by t1<=76.
- The value of i is not needed after the loop terminates, so remove i from blocks B1, B2, and B3 entirely by replacing it by t1.

5.6.2 Code Motion:

- An important modification that decreases the amount of code in a loop is code motion.
- Identify loop invariant computations and move them out of the block.
- A **loop-invariant** computation takes an expression that yields the same result independent of the number of times a loop is executed.
- In the example, statements (4) and (6) are loop invariant computations.

- Remove these two statements from B2 and place it before B2 as a separate block and name it as B3.

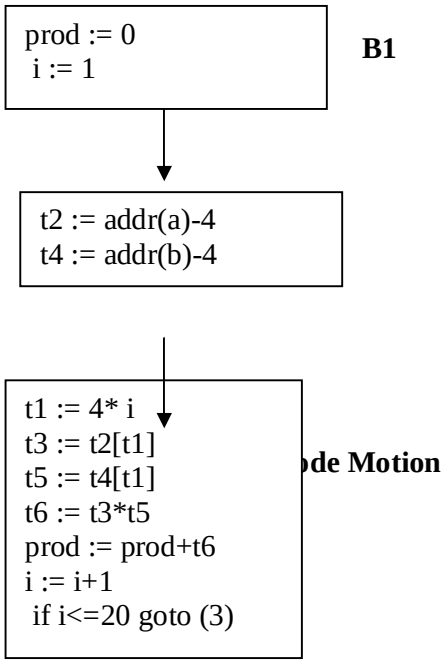


Fig 6:Flow graph for Code Motion

UNIT – V

Assignment-Cum-Tutorial Questions

Objective Questions

1.The intermediate codes that are in linear representation are _____ and _____.

2.The conversion of all labels in three address statements to addresses of instructions is known as _____.

3._____representation of three address code uses temporary variables.

4._____ representation makes the source language program independent.

5.One of the purposes of using intermediate code in compilers is to [ ]
a. Make parsing and semantic analysis simpler.
b. Improve error recovery and error reporting.
c. Increase the chances of reusing the machine-independent code optimizer in other compilers.
d. Improve the register allocation.

6.Some code optimizations are carried out on the intermediate code because [ ]
a. they enhance the portability of the compiler to other target processors
b. program analysis is more accurate on intermediate code than on machine code
c. the information from dataflow analysis cannot otherwise be used for optimization
d. the information from the front end cannot otherwise be used for optimization

7.The optimization technique which is typically applied on loops is [ ]
a. Removal of invariant computation
b. Peephole optimization
c. Constant folding
d. All of these

8.The identification of common sub-expression and replacement of run-time computations by compile-time computations is [ ]
a. Local optimization
b. Loop optimization
c. Constant folding
d. data flow analysis

9.Dead-code elimination in machine code optimization refers to: [ ]
a. Removal of all labels.
b. Removal of values that never get used.
c. Removal of function which are not involved.
d. Removal of a module after its use.

10. Code optimization is responsibility of: [ ]
a. Application programmer
b. System programmer
c. Operating system
d. All of the above

11. The polish noatation of the expression x+y*z is [ ]
a. xyz*+ b. x+yz* c. xyz+* d. *+xyz

12. For a C program accessing X[i][j][k], the following intermediate code is generated by a compiler. Assume that the size of an integer is 32 bits and the size of a character is 8 bits. [ ]

t0 = i * 1024
t1 = j * 32
t2 = k * 4
t3 = t1 + t0
t4 = t3 + t2
t5 = X[t4]

Which one of the following statements about the source code for the C program is CORRECT?
a. X is declared as “int X[32][32][8]”
b. X is declared as “int X[4][1024][32]”
c. X is declared as “char X[4][32][8]”
d. X is declared as “char X[32][16][2]”

13. The three address code for the statement x+-y*(-y+z) is [ ]
a. t1=y, t2=t1+z, t3=t1*t2, t4=x+t3
b. t1=-y, t2=x+t1, t3=t1+z, t4=t2*t3
c. both a and b are valid
d. none

14. Consider the following C code segment.

```
for (i = 0, i<n; i++)
{
for (j=0; j<n; j++)
{
if (i%2)
{
x += (4*j + 5*i);
y += (7 + 4*j);
}
}
```

}

Which one of the following is false? [ ]

a. The code contains loop invariant computation
b. There is scope of common sub-expression elimination in this code
c. There is scope of strength reduction in this code
d. There is scope of dead code elimination in this code

15. Replacing the expression 2*3.14 by 6.28 is [ ]

a. Constant folding b. Induction variable
c. Strength reduction d. Code reduction

16. Consider the intermediate code given below.

```
i = 1
j = 1
t1 = 5 * i
t2 = t1+ j
t3 = 4 * t2
t4 = t3
a[t4] = – 1
j = j + 1
if j < = 5 goto (3)
i = i + 1
if i < 5 goto (2)
```

The number of nodes and edges in the control-flow-graph constructed for the above code, respectively, are [ ]

a. 5 and 7
b. 6 and 7
c. 5 and 5
d. 7 and 8

17. Consider the basic block given below. [ ]

```
a = b + c
c = a + d
d = b + c
e = d - b
a = e + b
```

The minimum number of nodes and edges present in the DAG representation of the above basic block respectively are

a. 6 and 6
b. 8 and 10
c. 9 and 12
d. 4 and 4

SECTION-B

Descriptive Questions

1. What are the various types of intermediate code representation?
2. What are the applications of DAG?
3. Explain in detail about loop optimization techniques.
4. Explain in detail the procedure that eliminates global common sub expression with an example.
5. What is flow graph? Write an algorithm to partition a sequence of three-address statements into basic blocks
6. Explain about dead code elimination, constant folding and copy propagation with an example.

Problems

1. Convert the given infix expression into postfix expression.

   (a + b) *(c + d) (a + b + c)

2. Translate the expression –(a+b)*(c+d)+(a+b+c) into three address code , quadruples, triples and indirect triples.
3. Generate the three address code for an expression x: = a + b * c +d ;
4. Draw the DAG for the statement a=(a*b+c)-(a*b+c).
5. Construct the DAG for the following basic block

a= b + c
b = a - d
c= b + c
d= a – d

6. Write intermediate code for the following source code and identify basic blocks.

```
fact(x)
{
        int   f=1;
        for(i=2;i<=x;i++)
                f=f*i;
        return   f;
}
```

7. Optimize the following code

```
while( i<5000)
{
        A=sin(x)/cos(x)
}
```

8. Write intermediate code for the following source code: for i from 1 to 10 do for j from 1 to 10 do a[i, j]=0.0; for i from 1 to 10 do a[i, i]=1.0; and identify basic blocks.

GATE/NET/SLET

1. Consider the following code segment.

```
x = u - t;
y = x * v;
x = y + w;
y = t - z;
y = x * y;
```

The minimum number of total variables required to convert the above code segment to static *single assignment* form is_____.

2. The least number of temporary variables required to create a three-address code in static single assignment form for the expression q+r/3+s-t*5+u*v/w is _____.

3. Which one of the following is FALSE? [ ]

a. A basic block is a sequence of instructions where control enters the sequence at the beginning and exits at the end.
b. Available expression analysis can be used for common subexpression elimination.
c. Live variable analysis can be used for dead code elimination.
d. x = 4 □ 5 □ x = 20 is an example of common subexpression elimination.