**Experiment No:-09**

**Aim:** To study & implement Code Optimization Algorithm.

**Theory:**

**Code Optimization:**

Optimization is the process of transforming a piece of code to make more efficient (either in terms of time or space) without changing its output or side effects. The only difference visible to the code’s user should be that it runs faster and/or consumes less memory. It is really a misnomer that the name implies you are finding an "optimal" solution— in truth, optimization aims to improve, not perfect, the result. Optimization is the field where most compiler research is done today. The tasks of the front end (scanning, parsing, semantic analysis) are well understood and optimized code generation is relatively straightforward. Optimization, on the other hand, still retains a sizable measure of mysticism. Hi quality optimization is more of an art than a science. Compilers for mature languages aren’t judged by how well they parse or analyze the code—you just expect it to do it right with a minimum of hassle—but instead by the quality of the object code they produce. There are a variety of tactics for attacking optimization. Some techniques are applied to the intermediate code, to streamline, rearrange, compress, etc. in an effort to reduce the size of the abstract syntax tree or shrink the number of TAC instructions. Others are applied as part of final code generation—choosing which instructions to emit, how to allocate registers and when/what to spill, and the like. And still other optimizations may occur after final code generation, attempting to rework the assembly code itself into something more efficient.

**Control Flow Analysis:**

Consider all that has happened up to this point in the compiling process—lexical analysis, syntactic analysis, semantic analysis and finally intermediate code generation. The compiler has done an enormous amount of analysis, but it still doesn’t really know how the program does what it does. In control flow analysis, the compiler figures out even more information about how the program does its work, only now it can assume that there are no syntactic or semantic errors in the code.

**Local Optimizations:**

Optimizations performed exclusively within a basic block are called "local optimizations". These are typically the easiest to perform since we do not consider any control flow information, we just work with the statements within the block. Many of the local optimizations we will discuss have corresponding global optimizations that operate on the same principle, but require additional analysis to perform. We'll consider some of the more common local optimizations as examples.

**Constant Folding:**

Constant folding refers to the evaluation at compile time of expressions whose operands are known to be constant. In its simplest form, it involves determining that all of the operands in an expression are constant valued, performing the evaluation of the expression at compile time, and then replacing the expression by its value. If an expression such as

10+2\*3

is encountered, the compiler can compute the result at

Compile time (16) and emit code as if the input contained the result rather than the

original expression. Similarly, constant conditions, such as a conditional branch

if a<b

goto

L1

elsegoto L2

where a and b are constant can be replaced by a

Goto

L1 or Goto L2 depending on the truth of the expression evaluated at compile time.

**Constant Propagation:**

If a variable is assigned a constant value, then subsequent uses of that variable can be replaced by the constant as long as no intervening assignment has changed the value of the variable. Consider this section from our earlier Fibonacci example. On the left is the original, on the right is the improved version after constant propagation, which saves three instructions and removes the need for three temporary variables:

\_tmp4 = 0 ;

f0 = \_tmp4 ;

\_tmp5 = 1 ;

f1 = \_tmp5 ;

\_tmp6 = 2 ;

i = \_tmp6 ;

f0 = 0 ;

f1 = 1 ;

i = 2 ;

Algebraic Simplification And reassociation Simplifications use algebraic properties or

particular operator operand combinations to simplify expressions. Reassociation refers to using properties such as associativity,commutativity and distributivity to rearrange an expression to enable other optimizations such as constant folding or loop invariant code motion.

The most obvious of these are the optimizations that can remove useless instructions entirely via algebraic identities. The rules of arithmetic can come in handy when

looking for redundant calculations to eliminate. Consider the examples below, which allow you to replace an expression on the left with a simpler equivalent on the right:

x+0 = x

0+x = x

x\*1 = x

1\*x = x

0/x = 0

x0 = x

b&& true = b

b&& false = false

b || true = true

b || false = b

**Operator Strength Reduction:**

Operator strength reduction replaces an operator by a "less expensive" one. Given each group of identities below, which operations are the most and least expensive, assuming

f is a float and

i is an int

i\*2 = 2\*i = i+i = i<< 1

i/2 = (int)(i\*0.5)

01 = i

f\*2 = 2.0 \* f = f + f

f/2.0 = f\*0.5

Copy Propagation

This optimization is similar to constant propagation, but generalized to non constant values. If we have an assignment

a=b

in our instruction stream, we can replace later occurrences of a with b

Given the way we generate TAC code, this is a particularly valuable optimization since it is able to eliminate a large number of instructions that only serve to copy values from one variable to another.

The code on the left makes a copy of

tmp1 in tmp2 and a copy of tmp3 in tmp4

In the optimized version on the right, we eliminated those unnecessary copies and propagated the original variable into the later uses:

tmp2 = tmp1 ;

tmp3 = tmp2 \* tmp1;

tmp4 = tmp3 ;

tmp5 = tmp3 \* tmp2 ;

c = tmp5 + tmp4 ;

tmp3 = tmp1 \* tmp1 ;

tmp5 = tmp3 \* tmp1 ;

c = tmp5 + tmp3 ;

**Dead Code Elimination:**

If an instruction’s result is never used, the instruction is considered "dead" and can be removed from the instruction stream. So if we have

tmp1 = tmp2 + tmp3 ;

and tmp1

is never used again, we can eliminate this instruction altogether.

Common Subexpression Elimination

Two operations are common if they produce the same result. In such a case, it is likely more efficient to compute the result once and reference it the second time rather than reevaluate it. An expression is alive

if the operands used to compute the expression have not been changed. An expression that is no longer alive is dead.

tmp2 = x ;

x = 21 \* tmp2 ;

tmp3 = x \* x ;

tmp4 = x / y ;

y = tmp3 + tmp4 ;

tmp5 = x / y ;

z = tmp5 / tmp3 ;

y = z ;

**Global Optimizations, DataFlow Analysis:**

So far we were only considering making changes within one basic block. With some additional analysis, we can apply similar optimizations across basic blocks, making them global optimizations. It’s worth pointing out that global in this case does not mean across the entire program. We usually only optimize one function at a time. Inter procedural analysis is an even larger task, one not even attempted by some compilers.The additional analysis the optimizer must do to perform optimizations across basic blocks is called dataflow analysis. Dataflow analysis is much more complicated than control flow analysis, and we can only scratch the surface here, but were you to take CS243 (a wonderful class!) you will get to delve much deeper.

**Code Motion:**

Code motion (also called code hoisting) unifies sequences of code common to one or more basic blocks to reduce code size and potentially avoid expensive reevaluation. The most common form of code motion is loop invariant code motion that moves statements that evaluate to the same value every iteration of the loop to somewhere outside the loop. What statements inside the following

TAC code can be moved outside the loop body?

L0:

tmp1 = tmp2 + tmp3 ;

tmp4 = tmp4 + 1 ;

PushPramtmp4 ;

LCall \_PrintInt ;

PopParams 4;

tmp6 = 10 ;

tmp5 = tmp4 == tmp6 ;

IfZ tmp5 Goto L0 ;

**Program:**

import java.io.\*;

class TAC

{

private static final char[][] precedence = {

{'/', '1'},

{'\*', '1'},

{'+', '2'},

{'-', '2'}

};

private static int precedenceOf(String t)

{

char token = t.charAt(0);

for (int i=0; i < precedence.length; i++)

{

if (token == precedence[i][0])

{

return Integer.parseInt(precedence[i][1]+"");

}

}

return -1;

}

public static void main(String[] args) throws Exception

{

int i, j, opc=0;

char token;

boolean processed[];

String[][] operators = new String[10][2];

String expr="", temp;

BufferedReader in = new BufferedReader(new InputStreamReader(System.in));

System.out.print("\nEnter an expression: ");

expr = in.readLine();

processed = new boolean[expr.length()];

for (i=0; i < processed.length; i++)

{

processed[i] = false;

}

for (i=0; i < expr.length(); i++)

{

token = expr.charAt(i);

for (j=0; j < precedence.length; j++)

{

if (token==precedence[j][0])

{

operators[opc][0] = token+"";

operators[opc][1] = i+"";

opc++;

break;

}

}

}

System.out.println("\nOperators:\nOperator\tLocation");

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

{

System.out.println(operators[i][0] + "\t\t" + operators[i][1]);

}

//sort

for (i=opc-1; i >= 0; i--)

{

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

{

if (precedenceOf(operators[j][0]) > precedenceOf(operators[j+1][0]))

{

temp = operators[j][0];

operators[j][0] = operators[j+1][0];

operators[j+1][0] = temp;

temp = operators[j][1];

operators[j][1] = operators[j+1][1];

operators[j+1][1] = temp;

}

}

}

System.out.println("\nOperators sorted in their precedence:\nOperator\tLocation");

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

{

System.out.println(operators[i][0] + "\t\t" + operators[i][1]);

}

System.out.println();

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

{

j = Integer.parseInt(operators[i][1]+"");

String op1="", op2="";

if (processed[j-1]==true)

{

if (precedenceOf(operators[i-1][0]) == precedenceOf(operators[i][0]))

{

op1 = "t"+i;

}

else

{

for (int x=0; x < opc; x++)

{

if ((j-2) == Integer.parseInt(operators[x][1]))

{

op1 = "t"+(x+1)+"";

}

}

}

}

else

{

op1 = expr.charAt(j-1)+"";

}

if (processed[j+1]==true)

{

for (int x=0; x < opc; x++)

{

if ((j+2) == Integer.parseInt(operators[x][1]))

{

op2 = "t"+(x+1)+"";

}

}

}

else

{

op2 = expr.charAt(j+1)+"";

}

System.out.println("t"+(i+1)+" = "+op1+operators[i][0]+op2);

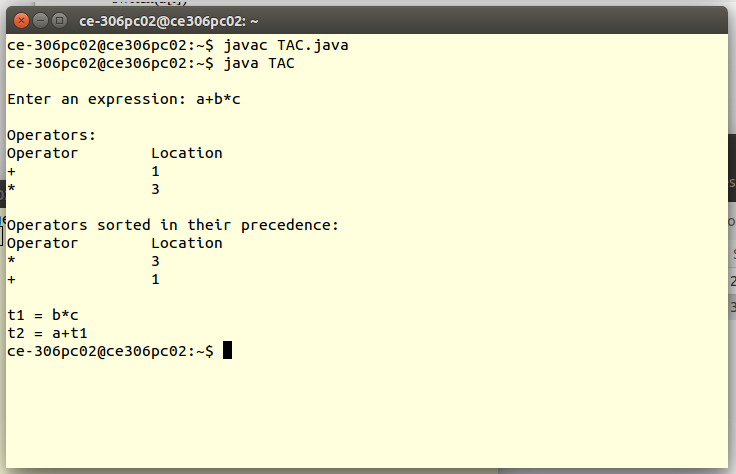
processed[j] = processed[j-1] = processed[j+1] = true;

}

}

}

**Output:**

****

**Conclusion:** Hence, we studied code optimization.