THEORY OF COMPUTATION AND COMPILER DESIGN

CSE2004

GROUP PROJECT

MODULE 5:
THREE ADDRESS CODES
CODE OPTIMIZATION TECHNIQUES
CODE GENERATION

PROBLEMS: CONVERSION FROM PARSE TREE TO TAC
OPTIMIZATION TECHNIQUES
CODE GENERATION

GROUP MEMBERS

ANURAG PRASAD	21BSA10075
MAYANK CHOUBEY	21BSA10113
RAJAT SHARMA	21BSA10119
VANSH THAKUR	21BSA10133

TABLE OF CONTENTS

01 THREE ADDRESS CODES

02 CODE OPTIMIZATION TECHNIQUES

03 CODE GENERATION

04 **PROBLEMS:** CONVERSION FROM PARSE TREE TO TAC-OPTIMIZATION TECHNIQUES -CODE GENERATION

THREE ADDRESS CODES

- Three-address code, often known as TAC or 3AC, is an intermediate code in computer science that is used by optimizing compilers to help in the implementation of code-improving transformations.
- At most, three operands can be used in a TAC instruction, which is normally made up of an assignment and a binary operator.
- Here's a brief explanation of the components of Three Address Code:
- Opcode: This is the operation code of the instruction, representing the operation to be performed. Examples include ADD, SUB, MUL, DIV, etc.
- Operand1, Operand2, and Operand3: These are the operands of the instruction. In a three-address instruction, there are at most three operands. These operands can be variables, constants, or temporary values.

THREE ADDRESS CODES

- Imagine you have a recipe for making a cake. The recipe tells you what ingredients you need and what steps to follow, but it doesn't actually make the cake for you. You still need to follow the instructions and do the work yourself.
- TAC is like a recipe for a computer program. It tells the computer what operations to perform, but it doesn't actually run the program itself. The computer still needs to follow the instructions and do the calculations.
- TAC is made up of instructions, and each instruction has three parts:

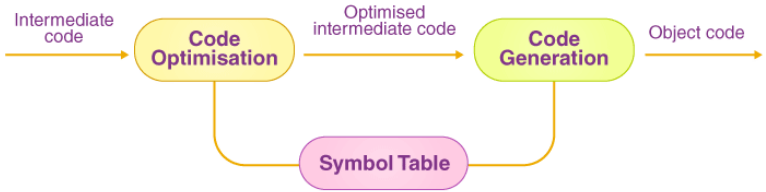
  The destination: This is where the result of the instruction will be stored.
  The operation: This is what the computer should do, such as adding two numbers or comparing two values.
  The source: This is where the data for the operation comes from.

- For example, the TAC instruction x = y + z means that the computer should add the values of y and z, and store the result in x.
- TAC is used in compiler design because it is a simple and efficient way to represent computer programs. It is also easy for computers to understand and execute.

CODE OPTIMIZATION IN COMPILER DESIGN

The code optimization in the synthesis phase is a program transformation technique, which tries to improve the intermediate code by making it consume fewer resources (i.e. CPU, Memory) so that faster-running machine code will result. Compiler optimizing process should meet the following objectives :

- The optimization must be correct, it must not, in any way, change the meaning of the program.
- Optimization should increase the speed and performance of the program.
- The compilation time must be kept reasonable.
- The optimization process should not delay the overall compiling process.



CODE OPTIMIZATION TECHNIQUES

Types of Code Optimization

The code optimization process can be broadly classified into two types :
Machine Independent Optimization
Machine Dependent Optimization

1. Machine Independent Optimization

This step of code optimization aims to optimize the intermediate code to produce a better target code. No CPU registers or absolute memory addresses are involved in the section of the intermediate code that is translated here.

2. Machine Dependent Optimization

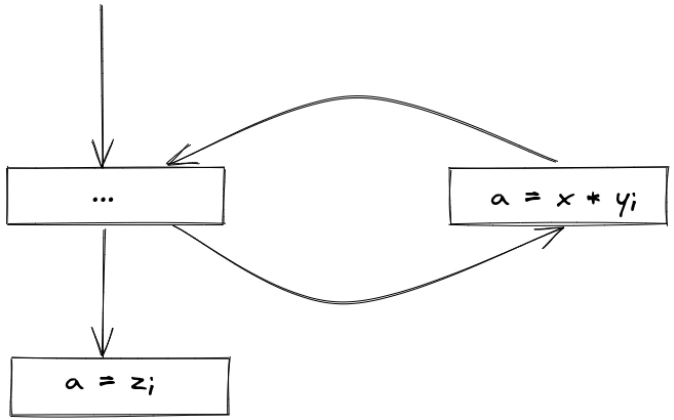
After the target code has been created and converted to fit the target machine architecture, machine-dependent optimization is performed. It may use absolute memory references rather than relative memory accesses and requires CPU registers. Machine-dependent optimizers make a concerted attempt to maximize the memory hierarchy's benefits.

CODE OPTIMIZATION TECHNIQUES

LOOP OPTIMIZATION

The majority of programs in the system operate in a loop. It is vital to optimize the loops to save CPU cycles and memory. The following strategies can be used to improve loops.

- **Loop-invariant code:** It is a piece of code that sits in the loop and computes the same value each time an iteration is performed. This code may be moved out of the loop by storing it to be calculated just once rather than with each iteration.
- **Induction analysis**: If a loop-invariant value changes the value of a variable within the loop, it is termed an induction variable.
- **Strength reduction**: Some expressions use more CPU cycles, time, and memory than others. These expressions should be replaced with less expensive expressions without sacrificing the expression's output. For example, multiplication (x * 2) uses more CPU cycles than (x 1) but produces the same output.

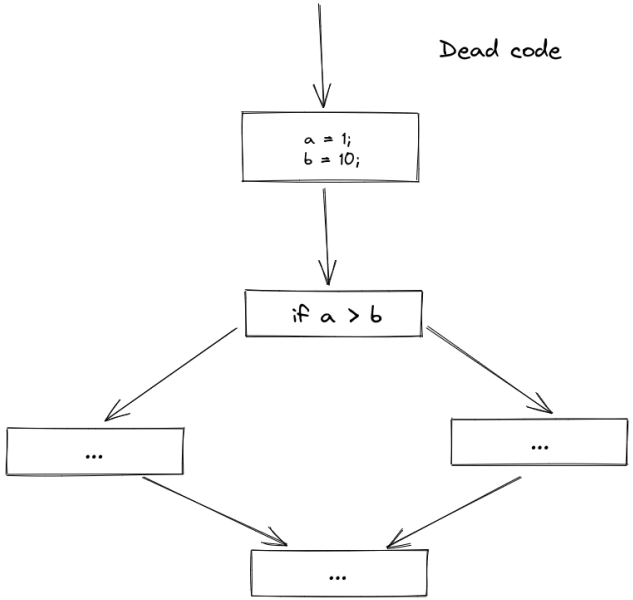


CODE OPTIMIZATION TECHNIQUES

PARTIALLY DEAD CODE

- Some code statements include calculated values utilized only in particular conditions, i.e., the values are used sometimes and not others. Partially dead-code refers to such codes.

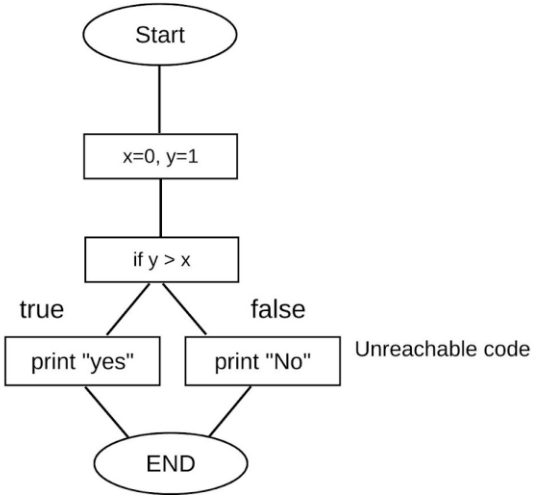
- The control flow diagram below shows a program section in which the variable 'a is utilized to assign the output of the equation 'x * y'. Let's pretend that the ‘a variable's value is never utilized within the loop. 'a' is given the variable 'z' value, which will be utilized later in the program, immediately after the control leaves the loop. We may infer that because the assignment code 'a' is never utilized anywhere, it is suitable for deletion.



CODE OPTIMIZATION TECHNIQUES

UNREACHABLE CODE ELIMINATION

A control flow graph should be created first. An inaccessible code block does not have an incoming edge. The inaccessible branches can be deleted after continual propagation and folding.



CODE OPTIMIZATION TECHNIQUES

FUNCTION INLINING

The body of the function takes the place of a function call. This saves a lot of time by eliminating the need to copy all parameters, store the return address, and so on. Let us explain this with an example below:

```
int addtwonum(int a, int b){
        return a+b;
}
int subtract(int a, int b){
        return addtwonum(a, -b);
}
```

Here, we see that by negating one of the numbers according to the context of the scenario, we call the addition function for subtraction. Now, let us see the below snippet:

```
int subtract(int a, int b){
return a+ (-b);
}
```

Here, we did the work of the function **addtwonum** itself into the subtract function. This is function inlining.

Other Important Manual Optimization

Here are some code optimization techniques with examples:

Compile-time evaluation: Compile-time evaluation is a technique that evaluates expressions at compile time instead of runtime. This can improve performance by eliminating redundant calculations. For example, the following code:

```
int a = 2 + 3;          |   int a = 5;
int b = a * 4;          |   int b = a*4;
```

(because the expression 2 + 3 can be evaluated at compile time to 5)

Variable propagation:-Variable propagation is a technique that propagates the values of variables throughout a program. This can eliminate redundant assignments and improve the accuracy of data flow analysis. For example,

```
int a = 5;                    | int a = 5;
int b = a;                    | int c = a+2;
int c = b + 2;                |
```

Because the value of a can propagated to b.

Dead code elimination:-

Dead code elimination is a technique that removes code that has no effect on the program's output. This can improve performance by reducing the amount of code that needs to be executed. For example

```
int a = 5;                     | int a =5;
int b = a + 2;                 |int b = a+2;
if (true) {                    | return a;
  return a;                    |
}
int c = b + 3;
```

Because the code assigns b and c is dead code.

Code motion :-

Code motion is a technique that moves code out of loops or from one place in a program to another. This can improve performance by reducing the number of times that the code is executed. For example

```
int sum = 0;                          | int sum = 0;
for (int i = 0; i < 10; ++i) {        |for(int i = 0 ; i<10 ; ++i){
  sum = sum + i;                      | sum += i;}
}
```

Because the code that assigns sum can be moved out of the loop.

Induction variable and strength reduction :-

Induction variable and strength reduction are techniques that are used to optimize loops. Induction variable optimization identifies and analyzes variables that are incremented or decremented by a constant value within a loop. Strength reduction replaces expensive operations with cheaper ones. For example,

```
int sum = 0;                          |int sum = 0;
for (int i = 0; i < 10; ++i) {        |int i = 0;
  sum = sum + i * i;                  |while(i<10){
}                                     |    sum += i*i;
                                      |     ++i;}
```

(because the code assigns sum can be reduced to simpler)

CODE GENERATION IN TOC

- In the realm of compiler design, code generation plays a crucial role, transforming an intermediate representation (IR) of a program into machine code, a language directly executable by the target machine. This final stage of the compilation process bridges the gap between the high-level abstraction of a programming language and the low-level instructions understood by the processor.

- Objectives of Code Generation:

- Correctness: The generated machine code must faithfully represent the semantics of the original source code.
- Efficiency: The generated machine code should execute efficiently, minimizing execution time and resource utilization.
- Optimality: The generated machine code should be optimized to the extent possible, considering factors such as instruction selection, register allocation, and instruction scheduling.

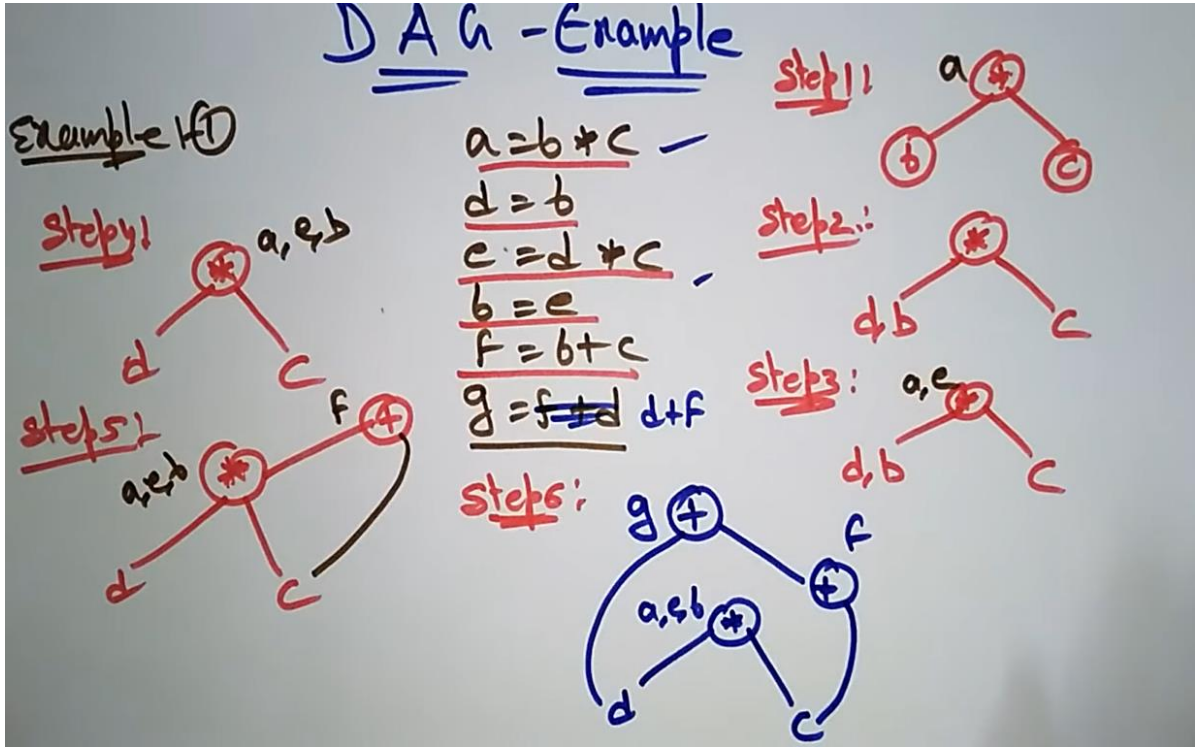
CODE GENERATION IN TOC

- Phases of Code Generation:

- Instruction Selection: Each IR instruction is mapped to an equivalent machine code instruction.
- Register Allocation: Register assignments are determined for variables and temporary values to maximize register usage.
- Instruction Scheduling: Machine code instructions are reordered to minimize instruction dependencies and pipeline stalls.
- Address Translation: Memory addresses are translated from virtual addresses to physical addresses.
- Code Optimization: Advanced optimizations are applied to further improve code efficiency.

PROBLEMS

DIRECTED ACYCLIC GRAPH (DAG) EXAMPLE



PROBLEMS

CONVERSION FROM PARSE TREE TO TAC

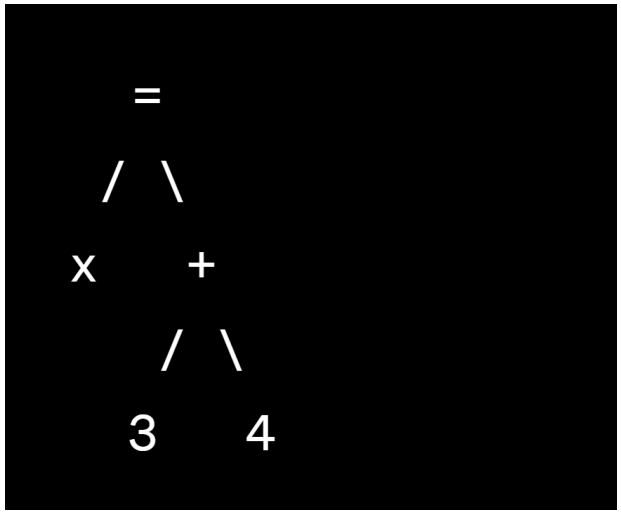
- The conversion from parse tree to TAC (three-address code) involves transforming a hierarchical representation of a program into a linear representation that can be more easily executed by a machine. The parse tree represents the syntax of the program, while the TAC represents the semantics of the program.

- The specific steps involved in the conversion will vary depending on the programming language and the compiler being used. However, the general process is as follows:

- Traversal: The parse tree is traversed in a depth-first or breadth-first manner.
- Code generation: At each node in the parse tree, code is generated to perform the operation represented by the node.
- Address assignment: Temporary variables are created for intermediate results, and addresses are assigned to these variables.
- Instruction ordering: The generated code is reordered to ensure that instructions are executed in the correct order.

PROBLEMS

CONVERSION FROM PARSE TREE TO TAC




```
t1 = 3 + 4
x = t1
```

PROBLEMS

DIRECTED ACYCLIC GRAPH (DAG)

$$x = (((a+a)_{t_1} + (a+a)_{t_2})_{t_3} + ((a+a)_{t_4} + (a+a)_{t_5})_{t_6})$$

$t_1 = a + a$

$t_2 = a + a$

$t_3 = t_1 + t_2$

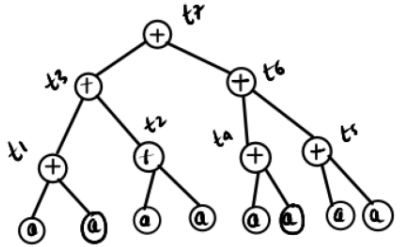
$t_4 = a + a$

$t_5 = a + a$

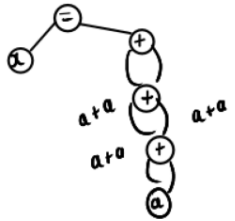
$t_6 = t_4 + t_5$

$t_7 = t_3 + t_6$

$x = t_7$



SYNTAX TREE



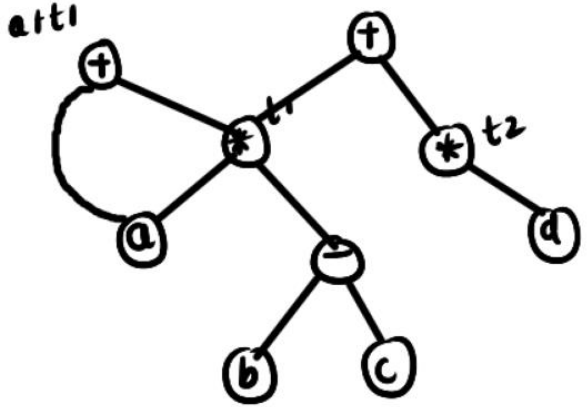
DAG

PROBLEMS

DIRECTED ACYCLIC GRAPH (DAG)

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

$a + t1 + t2$ ($t2 = b - c$)



PROBLEMS

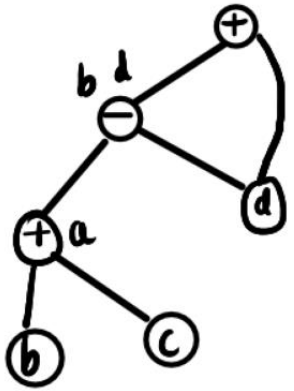
DIRECTED ACYCLIC GRAPH (DAG)

$a = b + c$

$b = a - d$

$c = b - c$

$d = a - d$



THANKS!