Experiment No.8
Implementation of code optimization phase of compiler.
Date of Performance: 24-03-2025
Date of Submission: 07-04-2025

Aim: Implementation of code optimization phase of compiler.

Objective: To develop an ability to implement code optimization techniques in design of compiler.

Theory:

Optimization is a program transformation technique, which tries to improve the code by making it consume less resources (i.e. CPU, Memory) and deliver high speed.

In optimization, high-level general programming constructs are replaced by very efficient low-level programming codes. A code optimizing process must follow the three rules given below:

- The output code must not, in any way, change the meaning of the program.
- Optimization should increase the speed of the program and if possible, the program should demand a smaller number of resources.
- Optimization should itself be fast and should not delay the overall compiling process.

Optimization can be categorized broadly into two types:

✔ Machine independent and

✔ Machine dependent.

Machine-independent Optimization

In this optimization, the compiler takes in the intermediate code and transforms a part of the code that does not involve any CPU registers and/or absolute memory locations. For example:

```
do
{
  item = 10;
  value = value + item;
} while(value<100);
```

This code involves repeated assignment of the identifier item, which if we put this way:

```
Item = 10;
do
{
  value = value + item;
} while(value<100);
```

should not only save the CPU cycles, but can be used on any processor.

Machine-dependent Optimization

Machine-dependent optimization is done after the target code has been generated and when the code is transformed according to the target machine architecture. It involves CPU registers and may have absolute memory references rather than relative references. Machine-dependent optimizers put efforts to take maximum advantage of memory hierarchy.

Basic Blocks

Source codes generally have a number of instructions, which are always executed in sequence and are considered as the basic blocks of the code. These basic blocks do not have any jump statements among them, i.e., when the first instruction is executed, all the instructions in the same basic block will be executed in their sequence of appearance without losing the flow control of the program.

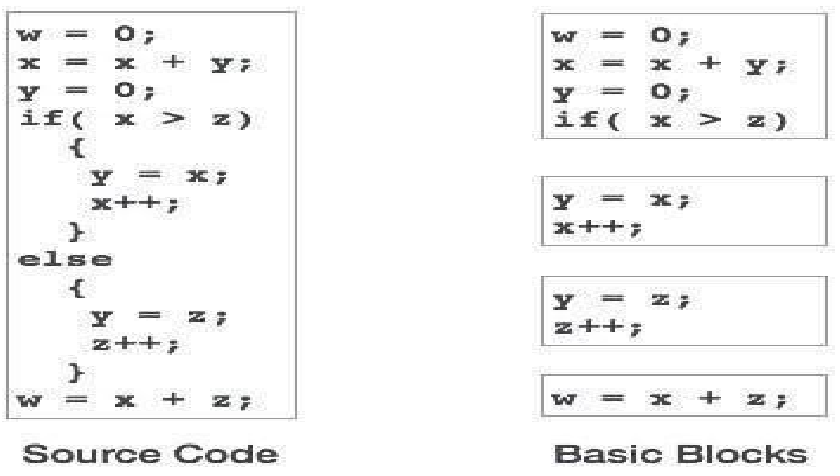
A program can have various constructs as basic blocks, like IF-THEN-ELSE, SWITCH-CASE conditional statements and loops such as DO-WHILE, FOR, and REPEAT-UNTIL, etc.

Basic block identification

We may use the following algorithm to find the basic blocks in a program:

- Search header statements of all the basic blocks from where a basic block starts:
    - First statement of a program.
    - Statements that are target of any branch (conditional/unconditional).
    - Statements that follow any branch statement.
- Header statements and the statements following them form a basic block.
- A basic block does not include any header statement of any other basic block.

Basic blocks are important concepts from both code generation and optimization point of view.



```
w = 0;
x = x + y;
y = 0;
if( x > z)
  {
   y = x;
   x++;
  }
else
  {
   y = z;
   z++;
  }
w = x + z;
```

Source Code

```
w = 0;
x = x + y;
y = 0;
if( x > z)
```

```
y = x;
x++;
```

```
y = z;
z++;
```

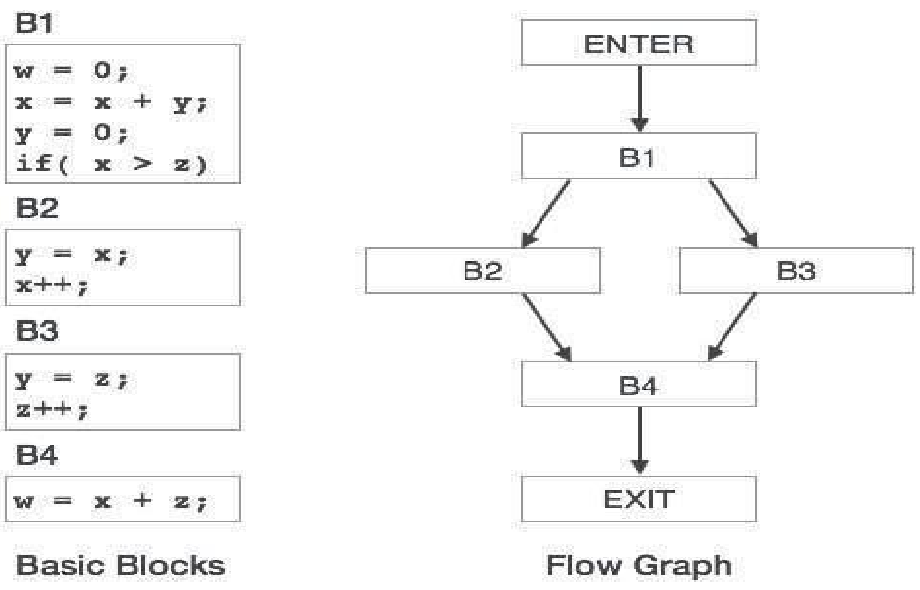
```
w = x + z;
```

Basic Blocks

Basic blocks play an important role in identifying variables, which are being used more than once in a single basic block. If any variable is being used more than once, the register memory allocated to that variable need not be emptied unless the block finishes execution.

Control Flow Graph

Basic blocks in a program can be represented by means of control flow graphs. A control flow graph depicts how the program control is being passed among the blocks. It is a useful tool that helps in optimization by help locating any unwanted loops in the program.


B1
```
w = 0;
x = x + y;
y = 0;
if( x > z)
```
B2
```
y = x;
x++;
```
B3
```
y = z;
z++;
```
B4
```
w = x + z;
```

Basic Blocks   Flow Graph

Loop Optimization

Most programs run as a loop in the system. It becomes necessary to optimize the loops in order to save CPU cycles and memory. Loops can be optimized by the following techniques:

- **Invariant code** : A fragment of code that resides in the loop and computes the same value at each iteration is called a loop-invariant code. This code can be moved out of the loop by saving it to be computed only once, rather than with each iteration.
- **Induction analysis** : A variable is called an induction variable if its value is altered within the loop by a loop-invariant value.
- **Strength reduction** : There are expressions that consume more CPU cycles, time, and memory. These expressions should be replaced with cheaper expressions without compromising the output of expression. For example, multiplication (x * 2) is expensive in terms of CPU cycles than (x << 1) and yields the same result.

Dead-code Elimination

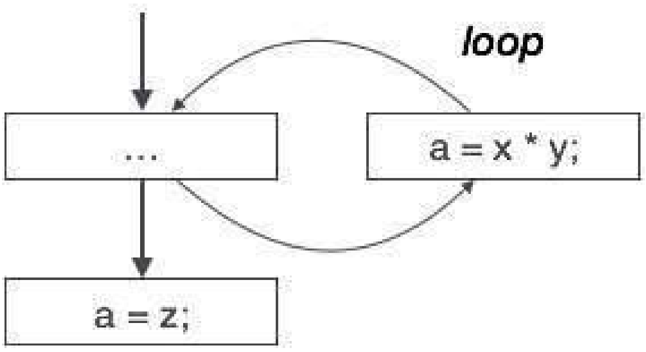
Dead code is one or more than one code statements, which are:

- Either never executed or unreachable,
- Or if executed, their output is never used.

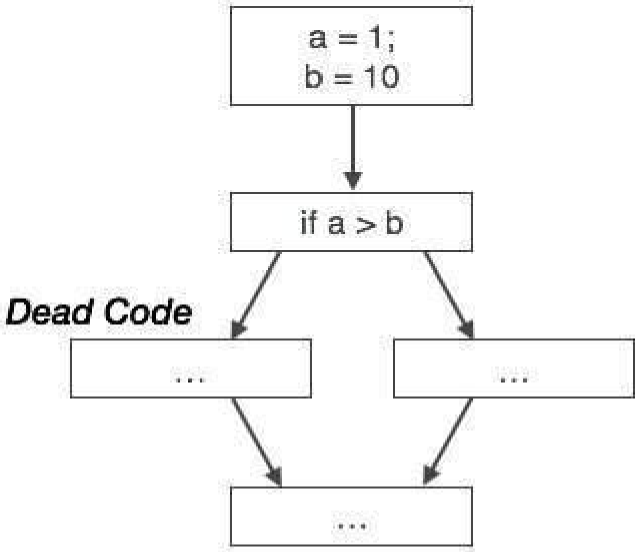
Thus, dead code plays no role in any program operation and therefore it can simply be eliminated.

Partially dead code

There are some code statements whose computed values are used only under certain circumstances, i.e., sometimes the values are used and sometimes they are not. Such codes are known as partially dead-code.



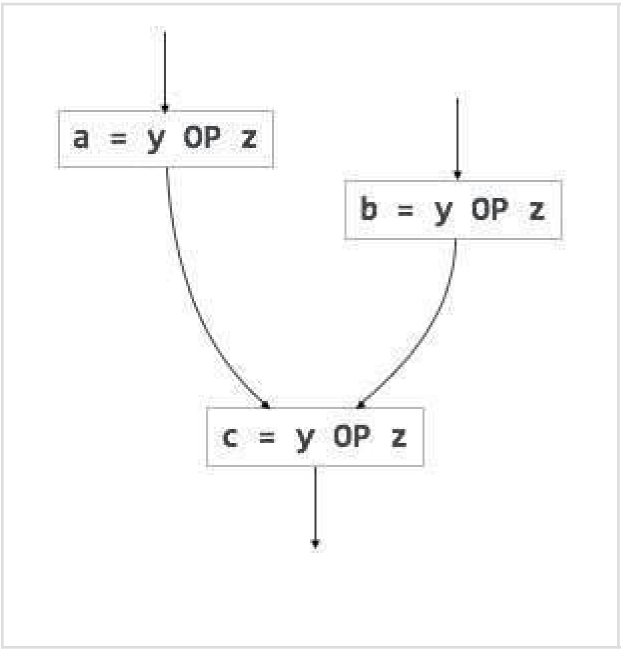
The above control flow graph depicts a chunk of program where variable 'a' is used to assign the output of expression 'x * y'. Let us assume that the value assigned to 'a' is never used inside the loop. Immediately after the control leaves the loop, 'a' is assigned the value of variable 'z', which would be used later in the program. We conclude here that the assignment code of 'a' is never used anywhere, therefore it is eligible to be eliminated.

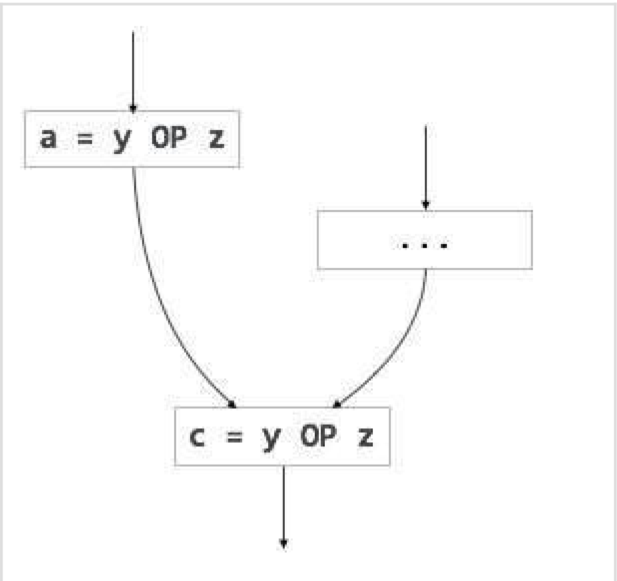


Likewise, the picture above depicts that the conditional statement is always false, implying that the code, written in true case, will never be executed, hence it can be removed.

Partial Redundancy

Redundant expressions are computed more than once in parallel path, without any change in operands.whereas partial-redundant expressions are computed more than once in a path, without any change in operands. For example,





[partially redundant expression]

[redundant expression]	

Loop-invariant code is partially redundant and can be eliminated by using a code-motion technique.

Another example of a partially redundant code can be:

```
If (condition)
{
  a = y OP z;
}
else
{
  ...
}
c = y OP z;
```

We assume that the values of operands (**y** and **z**) are not changed from assignment of variable **a** to variable **c**. Here, if the condition statement is true, then y OP z is computed twice, otherwise once.

Code motion can be used to eliminate this redundancy, as shown below:

```
If (condition)
{
  ...
  tmp = y OP z;
  a = tmp;
  ...
}
else
{
  ...
  tmp = y OP z;
}
c = tmp;
```

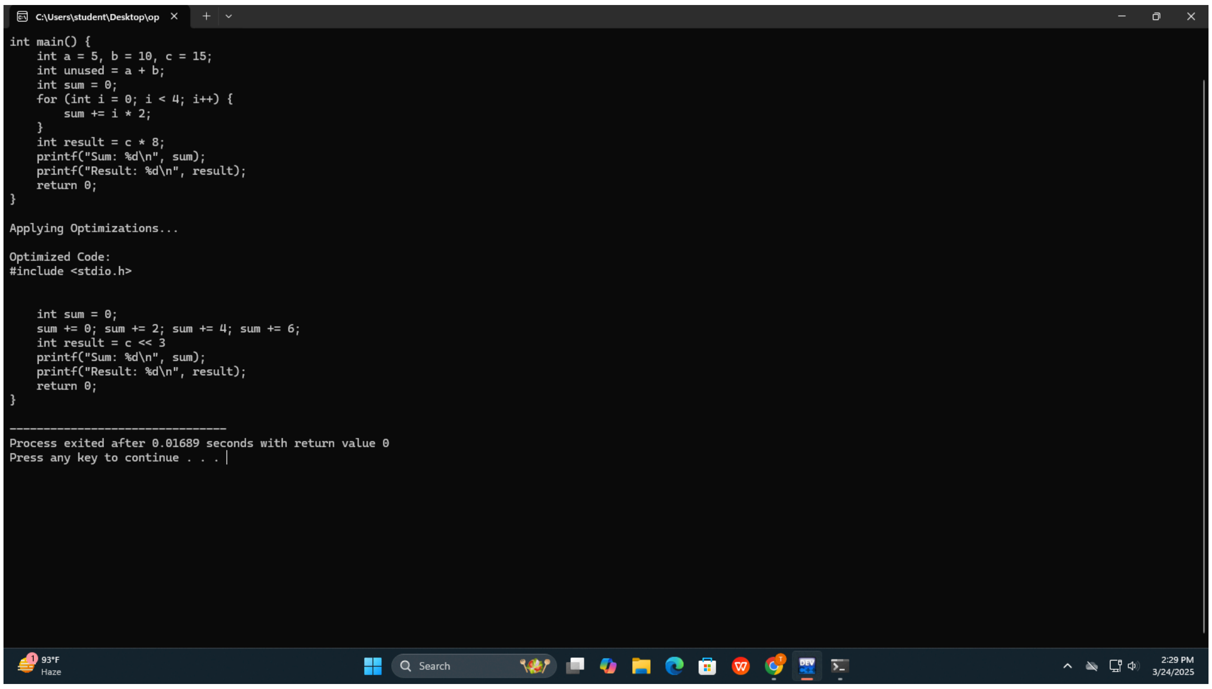
Here, whether the condition is true or false; y OP z should be computed only once.

Code:

```c
#include <stdio.h>

#include <string.h>

#include <stdlib.h>


// Function to count occurrences of a substring in a string

int countOccurrences(char *str, char *word) {

    int count = 0;

    char *pos = str;

    while ((pos = strstr(pos, word)) != NULL) {

        count++;

        pos += strlen(word);

    }

    return count;

}


// Function to remove dead code (variables used only once)

void removeDeadCode(char *code) {

    char *varStart = strstr(code, "int ");

    while (varStart) {

        char varName[50];

        if (sscanf(varStart, "int %s", varName) == 1) {

            // Remove potential trailing characters like ',' or ';'

            char *end = strpbrk(varName, ";,= ");

            if (end) *end = '\0';
            // Count occurrences of the variable in the entire code
            if (countOccurrences(code, varName) == 1) {
                // Remove the declaration line
                char *lineEnd = strstr(varStart, ";");
                if (lineEnd) {
                    memmove(varStart, lineEnd + 1, strlen(lineEnd + 1) + 1);
                }
            }
        }
        varStart = strstr(varStart + 1, "int "); // Find next variable declaration
    }
}

// Function to apply loop unrolling
void applyLoopUnrolling(char *code) {
    char *loopStart = strstr(code, "for (int i = 0; i < 4; i++)");
    if (loopStart) {
        char *loopEnd = strstr(loopStart, "}");
        if (loopEnd) {
            char unrolledLoop[] = "sum += 0; sum += 2; sum += 4; sum += 6;";
            memmove(loopStart, unrolledLoop, strlen(unrolledLoop));
            memmove(loopStart + strlen(unrolledLoop), loopEnd + 1, strlen(loopEnd + 1) + 1);
        }
    }
}
```

```
// Function to apply strength reduction
void applyStrengthReduction(char *code) {
    char *multiplyOp = strstr(code, "c * 8");
    if (multiplyOp) {
        memcpy(multiplyOp, "c << 3", 6); // Replace "c * 8" with "c << 3"
    }
}

// Function to optimize code
void optimizeCode(char *code) {
    printf("Original Code:\n%s\n", code);

    printf("\nApplying Optimizations...\n");

    // Apply optimizations
    removeDeadCode(code);
    applyLoopUnrolling(code);
    applyStrengthReduction(code);

    printf("\nOptimized Code:\n%s\n", code);
}

int main() {
    char inputCode[] =
        "#include <stdio.h>\n"
        "int main() {\n"
        "    int a = 5, b = 10, c = 15;\n"
        "    int unused = a + b;\n"
        "    int sum = 0;\n"
        "    for (int i = 0; i < 4; i++) {\n"
        "        sum += i * 2;\n"
        "    }\n"
        "    int result = c * 8;\n"
        "    printf(\"Sum: %d\\n\", sum);\n"
        "    printf(\"Result: %d\\n\", result);\n"
        "    return 0;\n"
        "}";

    optimizeCode(inputCode);
    return 0;
}
```

Output:



```
int main() {
    int a = 5, b = 10, c = 15;
    int unused = a + b;
    int sum = 0;
    for (int i = 0; i < 4; i++) {
        sum += i * 2;
    }
    int result = c * 8;
    printf("Sum: %d\n", sum);
    printf("Result: %d\n", result);
    return 0;
}

Applying Optimizations...

Optimized Code:
#include <stdio.h>


    int sum = 0;
    sum += 0; sum += 2; sum += 4; sum += 6;
    int result = c << 3
    printf("Sum: %d\n", sum);
    printf("Result: %d\n", result);
    return 0;
}

--------------------------------
Process exited after 0.01689 seconds with return value 0
Press any key to continue . . .
```

Conclusion:

Code optimization improves the efficiency and performance of the generated code without changing its output. In the given input, optimizations like **Dead Code Elimination** remove unused variables (e.g., `unused`), **Loop Optimization** simplifies repeated computations, and **Common Subexpression Elimination** reduces redundant calculations. The optimized code reduces memory usage and execution time, making it more efficient. This implementation highlights how applying optimization techniques enhances program execution and resource management, which is a crucial aspect of compiler design for generating high-performance, optimized machine-level code.