Experiment 02 : Code optimization Techniques

Learning Objective: Student should be able to Analyse and Apply code optimization techniques to increase efficiency of compiler.

Tools: Jdk1.8,Turbo C/C++, Python, Notepad++

Theory:

Code optimization aims at improving the execution efficiency. This is achieved in two ways.

1. Redundancies in a program are eliminated.

2. Computations in a program are rearranged or rewritten to make it execute efficiently.

The code optimization must not change the meaning of the program.

Constant Folding:

When all the operands in an operation are constants, operation can be performed at compilation time.

Elimination of common sub-expressions:

Common sub-expression are occurrences of expressions yielding the same value.

Implementation:

1. expressions with same value are identified
2. their equivalence is determined by considering whether their operands have the same values in all occurrences
3. Occurrences of sub-expression which satisfy the criterion mentioned earlier for expression can be eliminated.

Dead code elimination

Code which can be omitted from the program without affecting its result is called dead code. Dead code is detected by checking whether the value assigned in an assignment statement is used anywhere in the program

Frequency Reduction

Execution time of a program can be reduced by moving code from a part of program which is executed very frequently to another part of program, which is executed fewer times. For ex. Loop optimization moves, loop invariant code out of loop and places it prior to loop entry.

Strength reduction

The strength reduction optimization replaces the occurrence of a time consuming operations by an occurrence of a faster operation. For ex. Replacement of Multiplication by Addition

Example:

Before Optimization:	**After Optimization:**
A=B+C	A=B+C
B=A-D	B=A-D
C=B+C	C=B+C
D=A-D	D=B

Application: To optimize code for improving space and time complexity.

CODE:

Before Constant Folding:

```
t1 = 3
t2 = 7 * t1
t3 = t2 + 2
result = t3 * 5
```

After Constant Folding:

```
t1 = 3
t2 = 21
t3 = t2 + 2
result = t3 * 5
```

2. Elimination of common sub-expressions:

Before Common Sub-expressions:

```
t1 = 5 * 2
t2 = t1 + 3
t3 = t1 + 7
result = t2 * t3
```

After Common Sub-expressions:

```
t1 = 10
t2 = t1 + 3
t3 = t1 + 7
result = t2 * t3
```

3. Compile Time Evaluation:

Before Compile-Time Evaluation:

```
def calculate_square_and_double(n):
    t1 = n * n
    result = t1 * 2
```

After Compile-Time Evaluation:

```
t1 = 4
result = t1 * 2
```

3. Variable Propagation:

Before Variable Propagation:

```
t1 = 6
t2 = t1 * 3
t3 = t2 + 2
result = t3 + t1
```

After Variable Propagation:

```
t1 = 6
t2 = 18
t3 = t2 + 2
result = t3 + t1
```

Result and Discussion:

Learning Outcomes: The student should have the ability to

LO1: **Define** the role of Code Optimizer in Compiler design.
LO2: List the different principle sources of Code Optimization.
LO3: Apply different code optimization techniques for increasing efficiency of compiler.
LO4: Demonstrate the working of Code Optimizer in Compiler design.

Conclusion:

For Faculty Use

Correction Parameters	**Formative Assessment [40%]**	**Timely completion of Practical [40%]**	**Attendance / Learning Attitude [20%]**	
Marks Obtained				