**Shri Ramdeobaba College of Engineering and Management, Nagpur**

**Department of Computer Science and Engineering**

**Session: 2024-2025**

**Compiler Design Lab**



**PRACTICAL No. 6**

**Name – Siddhi Kothekar**

**Roll no. – 19**

**Batch – A2**

**Topic:** Code Optimization

**Aim:** Write a code to implement local optimization techniques until no further optimization is possible for the given three address code. **Input:** Three Address Code (non-optimized)

Original Three Address Code:

0: t0 = 4

1: t1 = 2

2: t2 = 4

3: t0 = 3 + t4

4: t1 = 3 - t2

5: t1 = 4 - 2

6: t0 = t2

7: t2 = 5 + t3

8: t3 = 9 / t4

9: t2 = 7 \* 8

10: t0 = 3 / 4

11: t1 = t4 / 9

12: t0 = 2 / 9

13: t3 = 10 \* 2

14: t4 = t3

**Implementation:** Identify and apply local optimization techniques to optimize the TAC

* Copy Propagation
* Constant propagation
* Constant Folding
* Common Subexpression Elimination
* Dead code elimination

Code:

import random import re from collections import defaultdict

class ThreeAddressCode:

def \_\_init\_\_(self): self.statements = [] self.variable\_map = {} *# For copy propagation* self.constant\_map = {} *# For constant propagation* self.expression\_map = {} *# For common subexpression elimination* self.used\_variables = set() *# For dead code elimination* self.defined\_variables = set() *# For dead code elimination*

def add\_statement(self, statement):

"""Add a statement to the three address code.""" self.statements.append(statement)

def generate\_random\_code(self, num\_statements=15, num\_variables=5):

"""Generate random three address code."""

operations = ['+', '-', '\*', '/'] variables = [f't{i}' for i in range(num\_variables)] constants = list(range(1, 11))

*# Generate initial assignments with constants to some variables* for i in range(min(num\_variables, 3)): var = variables[i] const = random.choice(constants) self.add\_statement(f"{var} = {const}")

*# Generate the rest of the statements* for \_ in range(num\_statements - min(num\_variables, 3)):

result = random.choice(variables)

*# Decide if it's a binary operation or a copy* if random.random() < 0.8: *# 80% chance for binary operation* op = random.choice(operations)

*# Choose operands (variables or constants)* operand1 = random.choice(variables + [str(c) for c in constants]) operand2 = random.choice(variables + [str(c) for c in constants])

self.add\_statement(f"{result} = {operand1} {op} {operand2}") else: *# 20% chance for a copy* source = random.choice(variables) if source != result: *# Avoid self-assignment* self.add\_statement(f"{result} = {source}")

return self.statements

def analyze\_code(self):

"""Analyze the code to prepare for optimizations.""" self.variable\_map = {} self.constant\_map = {} self.expression\_map = {} self.used\_variables = set() self.defined\_variables = set()

*# First pass: collect defined variables*

for stmt in self.statements: parts = stmt.split('=') if len(parts) == 2:

target = parts[0].strip() self.defined\_variables.add(target)

*# Second pass: collect used variables and build maps* for i, stmt in enumerate(self.statements): if '=' not in stmt:

continue

target, expression = [part.strip() for part in stmt.split('=', 1)]

*# For copy propagation (x = y)* if re.match(r'^\w+$', expression):

self.variable\_map[target] = expression

*# For constant propagation (x = constant)* if re.match(r'^\d+$', expression):

self.constant\_map[target] = int(expression)

*# For common subexpression elimination* tokens = re.findall(r'[+\-\*/]|\w+|\d+', expression) if len(tokens) == 3 and tokens[1] in ['+', '-', '\*', '/']: expr\_key = f"{tokens[0]} {tokens[1]} {tokens[2]}" if expr\_key not in self.expression\_map: self.expression\_map[expr\_key] = target

*# Collect used variables*

for token in re.findall(r'\b[a-zA-Z]\w\*\b', expression): self.used\_variables.add(token)

def copy\_propagation(self):

"""Perform copy propagation optimization.""" modified = False for i, stmt in enumerate(self.statements): if '=' not in stmt:

continue

target, expression = [part.strip() for part in stmt.split('=', 1)]

*# Replace variables with their copies* for var, val in self.variable\_map.items(): if var != target: *# Avoid cycles* pattern = r'\b' + re.escape(var) + r'\b' new\_expr = re.sub(pattern, val, expression) if new\_expr != expression:

self.statements[i] = f"{target} = {new\_expr}" modified = True expression = new\_expr

return modified

def constant\_propagation(self):

"""Perform constant propagation optimization.""" modified = False for i, stmt in enumerate(self.statements): if '=' not in stmt:

continue

target, expression = [part.strip() for part in stmt.split('=', 1)]

*# Replace variables with their constant values* for var, val in self.constant\_map.items(): if var != target: *# Avoid cycles* pattern = r'\b' + re.escape(var) + r'\b' new\_expr = re.sub(pattern, str(val), expression) if new\_expr != expression:

self.statements[i] = f"{target} = {new\_expr}" modified = True expression = new\_expr

return modified

def constant\_folding(self):

"""Perform constant folding optimization.""" modified = False for i, stmt in enumerate(self.statements): if '=' not in stmt:

continue

target, expression = [part.strip() for part in stmt.split('=', 1)]

*# Find expressions with constants only* *# Match patterns like "2 + 3", "5 \* 7", etc.*

match = re.search(r'(\d+)\s\*([+\-\*/])\s\*(\d+)', expression) if match:

operand1 = int(match.group(1)) operator = match.group(2) operand2 = int(match.group(3))

*# Compute the result* result = None if operator == '+':

result = operand1 + operand2 elif operator == '-':

result = operand1 - operand2 elif operator == '\*':

result = operand1 \* operand2 elif operator == '/' and operand2 != 0:

result = operand1 // operand2 *# Integer division*

if result is not None:

*# Replace the expression with the computed constant* new\_expr = expression.replace(match.group(0), str(result)) self.statements[i] = f"{target} = {new\_expr}"

*# If the entire right side is now a constant, update constant\_map* if new\_expr.isdigit():

self.constant\_map[target] = int(new\_expr)

modified = True

return modified

def common\_subexpression\_elimination(self):

"""Perform common subexpression elimination.""" modified = False for i, stmt in enumerate(self.statements): if '=' not in stmt:

continue

target, expression = [part.strip() for part in stmt.split('=', 1)]

*# Check if this is a binary operation* tokens = re.findall(r'[+\-\*/]|\w+|\d+', expression) if len(tokens) == 3 and tokens[1] in ['+', '-', '\*', '/']: expr\_key = f"{tokens[0]} {tokens[1]} {tokens[2]}"

*# If we've seen this expression before, replace it* if expr\_key in self.expression\_map and self.expression\_map[expr\_key] != target:

self.statements[i] = f"{target} = {self.expression\_map[expr\_key]}" modified = True

*# Add mapping for copy propagation* self.variable\_map[target] = self.expression\_map[expr\_key]

else:

*# Otherwise, record this expression* self.expression\_map[expr\_key] = target

return modified

def dead\_code\_elimination(self): """Perform dead code elimination.""" modified = False

*# First, update the analysis*

self.analyze\_code()

*# Find variables that are defined but never used* unused\_vars = self.defined\_variables - self.used\_variables

*# Remove statements that assign to unused variables*

i = 0 while i < len(self.statements): stmt = self.statements[i] if '=' in stmt:

target = stmt.split('=')[0].strip()

if target in unused\_vars and not re.search(r'\b' + re.escape(target) + r'\b', stmt.split('=')[1]):

self.statements.pop(i) modified = True continue i += 1

return modified

def optimize(self):

"""Apply all optimization techniques until no further optimization is possible.""" print("Original Three Address Code:") self.print\_code()

iteration = 1 while True:

self.analyze\_code()

*# Apply each optimization technique* cp\_mod = self.copy\_propagation() const\_prop\_mod = self.constant\_propagation() const\_fold\_mod = self.constant\_folding() cse\_mod = self.common\_subexpression\_elimination() dce\_mod = self.dead\_code\_elimination()

*# Check if any modifications were made* if cp\_mod or const\_prop\_mod or const\_fold\_mod or cse\_mod or dce\_mod:

print(f"\nAfter Iteration {iteration}:") self.print\_code() iteration += 1 else: break

print("\nFinal Optimized Three Address Code:") self.print\_code()

def print\_code(self):

"""Print the current three address code.""" for i, stmt in enumerate(self.statements):

print(f"{i}: {stmt}")

def main():

tac = ThreeAddressCode()

*# Option 1: Generate random TAC* tac.generate\_random\_code(num\_statements=15, num\_variables=5) *# Option 2: Use a predefined example (uncomment to use)*

""" tac.add\_statement("a = 3") tac.add\_statement("b = 4") tac.add\_statement("c = a + b") # c = 7 tac.add\_statement("d = b") # copy of b tac.add\_statement("e = c + d") # CSE opportunity tac.add\_statement("f = a + b") # CSE opportunity (same as c) tac.add\_statement("g = 2 \* f") # propagation opportunity tac.add\_statement("h = 3 + 4") # constant folding opportunity (h = 7) tac.add\_statement("i = h") # copy of h tac.add\_statement("j = 10") # dead code (j is never used)

"""

*# Apply optimizations* tac.optimize()

if \_\_name\_\_ == "\_\_main\_\_":

main()

**Output:** Optimized Three Address Code

After Iteration 1:

0: t0 = 4

1: t0 = 3 + t3

2: t0 = 4

3: t3 = 9 / t3

4: t0 = 0

5: t0 = 0

6: t3 = 20

After Iteration 2:

0: t3 = 9 / t3

1: t3 = 20

Final Optimized Three Address Code:

0: t3 = 9 / t3

1: t3 = 20