

**COMPILER CONSTRCTION**

**DELIVERABLE: 1+2+3**



**GROUP PARTNERS**

**22F-3851 ABDUL QUDOOS**

**22F-3263 SAMI SHAHZAD**

**22F-3363 SYED NOOR UL GHANI**

**LEXER.PY**

import ply.lex as lex

reserved = {

    'int': 'INT', 'float': 'FLOAT', 'bool': 'BOOL', 'string': 'STRING',

    'if': 'IF', 'else': 'ELSE', 'while': 'WHILE', 'for': 'FOR',

    'switch': 'SWITCH', 'case': 'CASE', 'print': 'PRINT', 'return': 'RETURN'

}

tokens = [

    'ID', 'NUMBER', 'STRING', 'PLUS', 'MINUS', 'TIMES', 'DIVIDE',

    'LPAREN', 'RPAREN', 'LBRACE', 'RBRACE', 'SEMICOLON', 'ASSIGN',

    'EQ', 'NEQ', 'LT', 'GT', 'LEQ', 'GEQ', 'AND', 'OR', 'NOT'

] + list(reserved.values())

# Token regex rules

t\_PLUS = r'\+'

t\_MINUS = r'-'

t\_TIMES = r'\\*'

t\_DIVIDE = r'/'

t\_LPAREN = r'\('

t\_RPAREN = r'\)'

t\_LBRACE = r'\{'

t\_RBRACE = r'\}'

t\_SEMICOLON = r';'

t\_ASSIGN = r'='

t\_EQ = r'=='

t\_NEQ = r'!='

t\_LT = r'<'

t\_GT = r'>'

t\_LEQ = r'<='

t\_GEQ = r'>='

t\_AND = r'&&'

t\_OR = r'\|\|'

t\_NOT = r'!'

t\_STRING = r'"([^"\n])\*"'

def t\_ID(t):

    r'[a-zA-Z\_][a-zA-Z0-9\_]\*'

    t.type = reserved.get(t.value, 'ID')

    return t

def t\_NUMBER(t):

    r'\d+(\.\d+)?'

    t.value = float(t.value) if '.' in t.value else int(t.value)

    return t

t\_ignore = ' \t'

def t\_newline(t):

    r'\n+'

    t.lexer.lineno += len(t.value)

# Handling comments properly

def t\_COMMENT(t):

    r'(/\\*[\s\S]\*?\\*/|//[^\n]\*)'

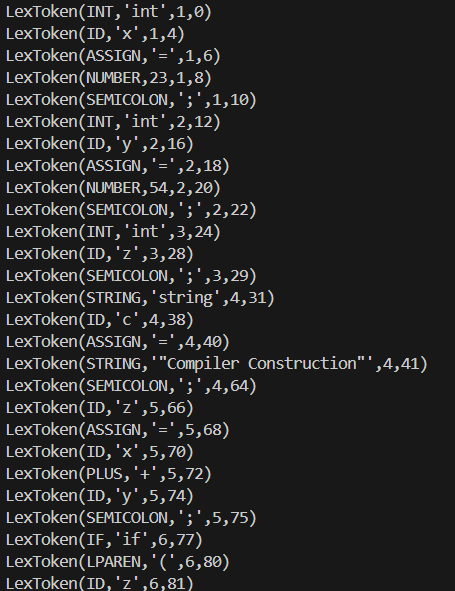
    pass

def t\_error(t):

    print(f"Illegal character '{t.value[0]}' at line {t.lexer.lineno}")

    t.lexer.skip(1)

lexer = lex.lex()



**SYNTAX.PY**

import ply.yacc as yacc

from lexer import tokens

# Symbol Table

symbol\_table = {}

precedence = (

    ('left', 'OR'),

    ('left', 'AND'),

    ('left', 'NOT'),

    ('left', 'GT', 'LT', 'GEQ', 'LEQ', 'EQ', 'NEQ'),

    ('left', 'PLUS', 'MINUS'),

    ('left', 'TIMES', 'DIVIDE'),

)

def p\_program(p):

    '''program : statement\_list'''

    p[0] = p[1]

def p\_statement\_list(p):

    '''statement\_list : statement

                      | statement\_list statement'''

    p[0] = [p[1]] if len(p) == 2 else p[1] + [p[2]]

def p\_statement\_declaration(p):

    '''statement : INT ID SEMICOLON

                 | FLOAT ID SEMICOLON

                 | BOOL ID SEMICOLON

                 | STRING ID SEMICOLON

                 | INT ID ASSIGN expression SEMICOLON

                 | FLOAT ID ASSIGN expression SEMICOLON

                 | BOOL ID ASSIGN expression SEMICOLON

                 | STRING ID ASSIGN expression SEMICOLON'''

    var\_type = p[1]

    var\_name = p[2]

    if len(p) == 4:

        symbol\_table[var\_name] = {"type": var\_type, "value": None}

        p[0] = ("declaration", var\_type, var\_name)

    else:

        symbol\_table[var\_name] = {"type": var\_type, "value": p[4]}

        p[0] = ("declaration", var\_type, var\_name, p[4])

def p\_statement\_assignment(p):

    '''statement : ID ASSIGN expression SEMICOLON'''

    var\_name = p[1]

    if var\_name not in symbol\_table:

        print(f"Semantic Error: Variable '{var\_name}' is not declared.")

    else:

        symbol\_table[var\_name]["value"] = p[3]

    p[0] = ("assignment", var\_name, p[3])

# if-else etc

def p\_statement\_if\_else(p):

    '''statement : IF LPAREN expression RPAREN statement ELSE statement

                 | IF LPAREN expression RPAREN LBRACE statement\_list RBRACE ELSE LBRACE statement\_list RBRACE'''

    if len(p) == 8:

        p[0] = ("if\_else", p[3], [p[5]], [p[7]])

    else:

        p[0] = ("if\_else", p[3], p[6], p[10])

# --- Loops (While) ---

def p\_statement\_while(p):

    '''statement : WHILE LPAREN expression RPAREN statement

                 | WHILE LPAREN expression RPAREN LBRACE statement\_list RBRACE'''

    p[0] = ("while", p[3], [p[5]] if len(p) == 6 else p[6])

# --- Print Statement ---

def p\_statement\_print(p):

    '''statement : PRINT LPAREN expression RPAREN SEMICOLON'''

    p[0] = ("print", p[3])

# --- Expressions (Arithmetic, Relational, Boolean) ---

def p\_expression\_arithmetic(p):

    '''expression : expression PLUS expression

                  | expression MINUS expression

                  | expression TIMES expression

                  | expression DIVIDE expression'''

    p[0] = ("binary\_op", p[2], p[1], p[3])

def p\_expression\_relational(p):

    '''expression : expression GT expression

                  | expression LT expression

                  | expression GEQ expression

                  | expression LEQ expression

                  | expression EQ expression

                  | expression NEQ expression'''

    p[0] = ('relational\_op', p[2], p[1], p[3])

def p\_expression\_boolean(p):

    '''expression : expression AND expression

                  | expression OR expression

                  | NOT expression'''

    if len(p) == 3:

        p[0] = ("unary\_op", "NOT", p[2])

    else:

        p[0] = ("binary\_op", p[2], p[1], p[3])

def p\_expression\_number(p):

    '''expression : NUMBER'''

    p[0] = ("number", p[1])

def p\_expression\_identifier(p):

    '''expression : ID'''

    var\_name = p[1]

    if var\_name not in symbol\_table:

        print(f"Semantic Error: Variable '{var\_name}' is not declared.")

    p[0] = ("identifier", var\_name)

def p\_expression\_string(p):

    '''expression : STRING'''

    p[0] = ("string", p[1][1:-1])

def p\_error(p):

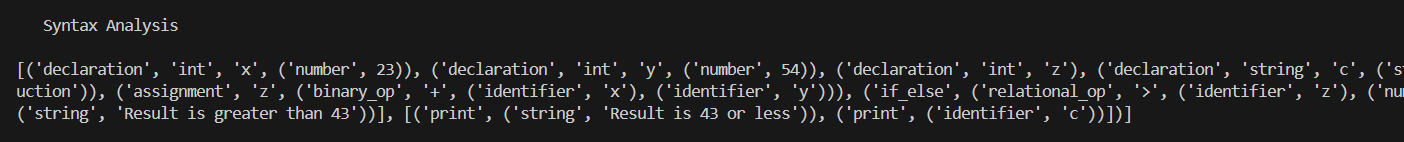
    if p:

        print(f"Syntax Error at '{p.value}' (Line {p.lineno})")

    else:

        print("Syntax Error at EOF")

parser = yacc.yacc()



**SEMANTIC.PY**

import syntax

def analyze(node):

    if isinstance(node, list):

        for stmt in node:

            analyze(stmt)

    elif isinstance(node, tuple):

        if node[0] == "declaration":

            symbol\_table[node[2]] = node[1]

            print(f"Declared: {node[2]} as {node[1]}")

        elif node[0] == "assignment":

            if node[1] not in symbol\_table:

                print(f"Semantic Error: Undeclared variable '{node[1]}'")

            else:

                print(f"Assigned value to {node[1]}")

        elif node[0] == "binary\_op":

            left = analyze(node[2])

            right = analyze(node[3])

            if left != right:

                print(f"Semantic Error: Type mismatch ({left} vs {right}) in '{node[1]}' operation")

            return left

        elif node[0] == "identifier":

            if node[1] not in symbol\_table:

                print(f"Semantic Error: Undeclared variable '{node[1]}'")

            return symbol\_table.get(node[1], None)

symbol\_table = {}

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

    ast = syntax.parse('int x = 10; print(x);')

    analyze(ast)

A screenshot of a computer screen

AI-generated content may be incorrect.

**Sample code:**

int x = 23;

int y = 54;

int z;

string c ="Compiler Construction";

z = x + y;

if (z > 43) {

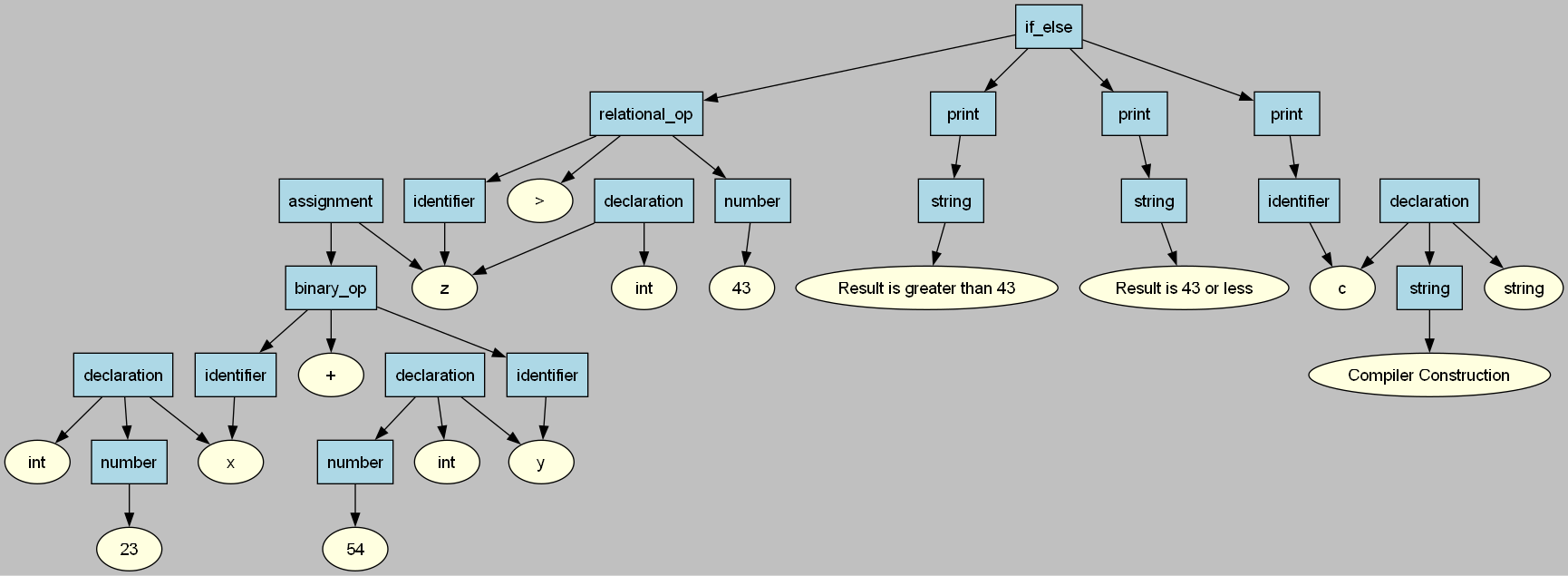
print("Result is greater than 43");

} else {

print("Result is 43 or less");

print(c);

}

**Abstract Syntax Tree (AST) :**

**IR.py**

temp\_count = 0

label\_count = 0

def new\_temp():

    global temp\_count

    temp\_count += 1

    return f"t{temp\_count}"

def new\_label():

    global label\_count

    label\_count += 1

    return f"L{label\_count}"

def generate\_expr\_ir(expr):

    if expr[0] == "number":

        return str(expr[1]), []

    elif expr[0] == "identifier":

        return expr[1], []

    elif expr[0] == "string":

        return expr[1], []

    elif expr[0] == "binary\_op":

        op = expr[1]

        left\_var, left\_ir = generate\_expr\_ir(expr[2])

        right\_var, right\_ir = generate\_expr\_ir(expr[3])

        temp = new\_temp()

        instr = (op, temp, left\_var, right\_var)

        return temp, left\_ir + right\_ir + [instr]

    else:

        raise ValueError(f"Unknown expression type: {expr[0]}")

def generate\_statement\_ir(stmt):

    if stmt[0] == "assignment":

        var = stmt[1]

        expr\_var, expr\_ir = generate\_expr\_ir(stmt[2])

        return expr\_ir + [("=", var, expr\_var)]

    elif stmt[0] == "if\_else":

        cond = stmt[1]

        then\_stmt = stmt[2]

        else\_stmt = stmt[3]

        cond\_var, cond\_ir = generate\_expr\_ir(cond)

        L1 = new\_label()

        L2 = new\_label()

        L\_loop\_start = new\_label()

        then\_ir = []

        if isinstance(then\_stmt, list):

            for s in then\_stmt:

                then\_ir.extend(generate\_statement\_ir(s))

        else:

            then\_ir = generate\_statement\_ir(then\_stmt)

        else\_ir = []

        if isinstance(else\_stmt, list):

            for s in else\_stmt:

                else\_ir.extend(generate\_statement\_ir(s))

        else:

            else\_ir = generate\_statement\_ir(else\_stmt)

        # Simulate a loop for the first if (i < 5)

        if cond[1] == "<":  # Treat as a loop

            ir = ([("label", L\_loop\_start)] +

                  cond\_ir +

                  [("ifnot", cond\_var, L1)] +

                  then\_ir +

                  [("goto", L\_loop\_start)] +

                  [("label", L1)] +

                  else\_ir +

                  [("label", L2)])

        else:  # Regular if-else

            ir = (cond\_ir +

                  [("ifnot", cond\_var, L1)] +

                  then\_ir +

                  [("goto", L2)] +

                  [("label", L1)] +

                  else\_ir +

                  [("label", L2)])

        return ir

    elif stmt[0] == "print":

        expr\_var, expr\_ir = generate\_expr\_ir(stmt[1])

        return expr\_ir + [("print", expr\_var)]

    elif stmt[0] == "declaration":

        var\_name = stmt[1]

        if len(stmt) > 2:

            expr\_var, expr\_ir = generate\_expr\_ir(stmt[2])

            return expr\_ir + [("=", var\_name, expr\_var)]

        return [("=", var\_name, 0)]

    else:

        raise ValueError(f"Unknown statement type: {stmt[0]}")

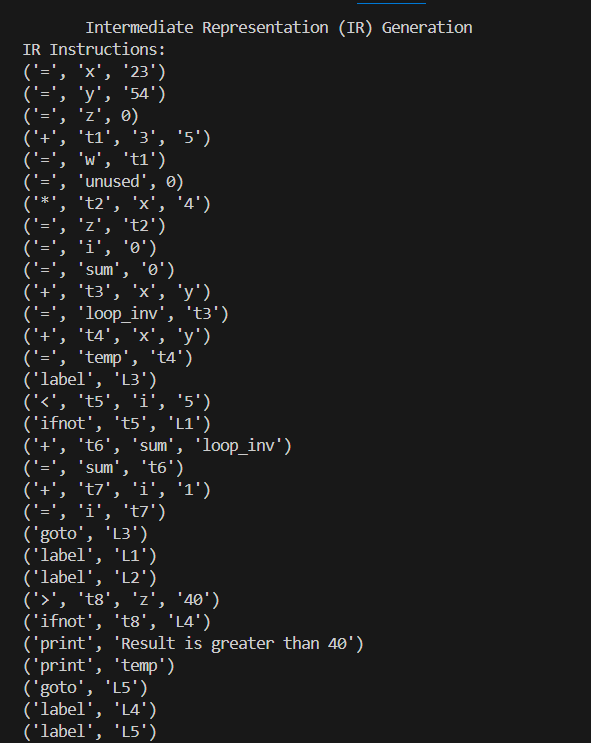
def generate\_ir(ast):

    ir = []

    for stmt in ast:

        ir.extend(generate\_statement\_ir(stmt))

    return ir



**OC.py**

def get\_used\_vars(ir):

    used\_vars = set()

    for instr in ir:

        if instr[0] in ("=", "+", "-", "\*", ">", "<", "ifnot", "print"):

            for i in range(2, len(instr)):

                if isinstance(instr[i], str) and not instr[i].startswith("L"):

                    used\_vars.add(instr[i])

    return used\_vars

def constant\_folding(ir):

    new\_ir = []

    for instr in ir:

        if instr[0] in ("+", "-", "\*", ">", "<"):

            op, result, left, right = instr

            try:

                left\_val = int(left)

                right\_val = int(right)

                if op == "+":

                    new\_ir.append(("=", result, str(left\_val + right\_val)))

                elif op == "-":

                    new\_ir.append(("=", result, str(left\_val - right\_val)))

                elif op == "\*":

                    new\_ir.append(("=", result, str(left\_val \* right\_val)))

                elif op == ">":

                    new\_ir.append(("=", result, str(int(left\_val > right\_val))))

                elif op == "<":

                    new\_ir.append(("=", result, str(int(left\_val < right\_val))))

            except ValueError:

                new\_ir.append(instr)

        else:

            new\_ir.append(instr)

    return new\_ir

def dead\_code\_elimination(ir):

    used\_vars = get\_used\_vars(ir)

    optimized\_ir = []

    for instr in ir:

        if instr[0] == "=" and instr[1] not in used\_vars:

            continue

        optimized\_ir.append(instr)

    return optimized\_ir

def strength\_reduction(ir):

    new\_ir = []

    for instr in ir:

        if instr[0] == "\*" and isinstance(instr[3], str):

            try:

                val = int(instr[3])

                if val == 2:

                    new\_ir.append(("<<", instr[1], instr[2], "1"))

                elif val == 4:

                    new\_ir.append(("<<", instr[1], instr[2], "2"))

                else:

                    new\_ir.append(instr)

            except ValueError:

                new\_ir.append(instr)

        else:

            new\_ir.append(instr)

    return new\_ir

def loop\_invariant\_code\_motion(ir):

    new\_ir = []

    loop\_start = None

    loop\_end = None

    invariants = []

    in\_loop = False

    for i, instr in enumerate(ir):

        if instr[0] == "label" and not in\_loop:

            loop\_start = instr[1]

            in\_loop = True

        elif instr[0] == "label" and in\_loop and i > 0 and ir[i-1][0] == "goto" and ir[i-1][1] == loop\_start:

            loop\_end = instr[1]

            in\_loop = False

        elif in\_loop and instr[0] in ("+", "-", "\*"):

            # Check if operands are loop-invariant (not modified in loop)

            op, result, left, right = instr

            modified\_vars = set()

            for j in range(i, len(ir)):

                if ir[j][0] == "=" and ir[j][1] in (left, right):

                    modified\_vars.add(ir[j][1])

                if ir[j][0] == "label" and ir[j][1] == loop\_end:

                    break

            if left not in modified\_vars and right not in modified\_vars:

                invariants.append(instr)

                continue

        new\_ir.append(instr)

    # Move invariants before loop

    if invariants and loop\_start:

        for i, instr in enumerate(new\_ir):

            if instr[0] == "label" and instr[1] == loop\_start:

                new\_ir[i:i] = invariants

                break

    return new\_ir

def common\_subexpression\_elimination(ir):

    new\_ir = []

    seen\_expressions = {}

    for instr in ir:

        if instr[0] in ("+", "-", "\*"):

            op, result, left, right = instr

            expr\_key = (op, left, right)

            if expr\_key in seen\_expressions:

                # Replace with the previous result

                new\_ir.append(("=", result, seen\_expressions[expr\_key]))

            else:

                seen\_expressions[expr\_key] = result

                new\_ir.append(instr)

        else:

            new\_ir.append(instr)

    return new\_ir

def optimize\_ir(ir):

    ir = constant\_folding(ir)

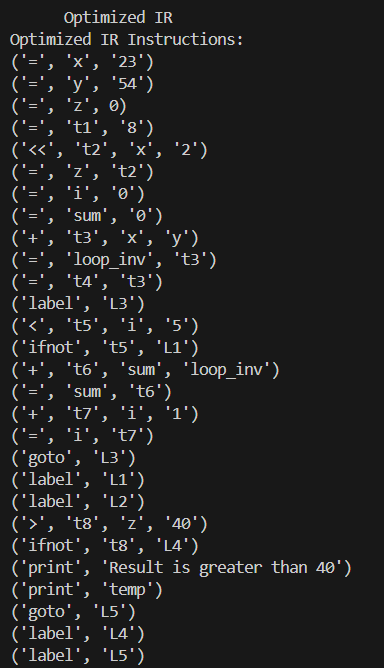
    ir = common\_subexpression\_elimination(ir)

    ir = loop\_invariant\_code\_motion(ir)

    ir = strength\_reduction(ir)

    ir = dead\_code\_elimination(ir)

    return ir



**Code\_gen.py:**

registers = ["eax", "ebx", "ecx", "edx"]

register\_map = {}

stack\_offset = 0

variable\_locations = {}

def allocate\_register(var):

    global stack\_offset

    if var in register\_map:

        return register\_map[var]

    for reg in registers:

        if reg not in register\_map.values():

            register\_map[var] = reg

            variable\_locations[var] = reg

            return reg

    stack\_offset -= 4

    variable\_locations[var] = f"dword [ebp{stack\_offset}]"

    return variable\_locations[var]

def free\_register(var):

    if var in register\_map:

        del register\_map[var]

    if var in variable\_locations:

        del variable\_locations[var]

def get\_location(var):

    if isinstance(var, int):

        return str(var)

    if var in variable\_locations:

        return variable\_locations[var]

    return allocate\_register(var)

def generate\_assembly(ir, filename="output.s"):

    global stack\_offset

    stack\_offset = 0

    register\_map.clear()

    variable\_locations.clear()

    with open(filename, "w") as f:

        # Header

        f.write("global \_main\n")

        f.write("extern \_printf\n")

        f.write("extern \_exit\n\n")

        # Data section

        f.write("section .data\n")

        f.write('    fmt\_int db "%d", 10, 0\n')

        f.write('    fmt\_str db "%s", 10, 0\n')

        f.write('    msg db "Operation result stored", 10, 0\n')

        f.write('    str1 db "Result is greater than 40", 10, 0\n\n')

        # BSS section

        f.write("section .bss\n")

        f.write("    res resb 4\n\n")

        # Text section

        f.write("section .text\n")

        f.write("\_main:\n")

        f.write("    push ebp\n")

        f.write("    mov ebp, esp\n")

        f.write(f"    sub esp, {abs(stack\_offset) + 64}\n\n")

        # Generate code from IR

        for instr in ir:

            asm = ir\_to\_asm(instr)

            if asm:

                for line in asm.split('\n'):

                    if line.strip():

                        f.write(f"    {line}\n")

        # Cleanup and exit

        f.write("\n    ; Program exit\n")

        f.write("    push msg\n")

        f.write("    push fmt\_str\n")

        f.write("    call \_printf\n")

        f.write("    add esp, 8\n")

        f.write("    mov esp, ebp\n")

        f.write("    pop ebp\n")

        f.write("    push 0\n")

        f.write("    call \_exit\n")

def ir\_to\_asm(instr):

    if not isinstance(instr, tuple):

        return f"; unhandled: {instr}"

    op = instr[0]

    if op == "=":

        dest, src = instr[1], instr[2]

        loc\_dest = get\_location(dest)

        if isinstance(src, int):

            if "dword [" in loc\_dest:

                return f"mov dword {loc\_dest}, {src}"

            return f"mov {loc\_dest}, {src}"

        else:

            loc\_src = get\_location(src)

            if "dword [" in loc\_dest and "dword [" in loc\_src:

                return f"mov eax, {loc\_src}\nmov dword {loc\_dest}, eax"

            elif "dword [" in loc\_dest:

                return f"mov dword {loc\_dest}, {loc\_src}"

            elif "dword [" in loc\_src:

                return f"mov eax, {loc\_src}\nmov {loc\_dest}, eax"

            return f"mov {loc\_dest}, {loc\_src}"

    elif op in ("+", "add"):

        dest, src1, src2 = instr[1], instr[2], instr[3]

        loc\_dest = get\_location(dest)

        loc\_src1 = get\_location(src1)

        loc\_src2 = get\_location(src2)

        asm = []

        # Load first operand

        if isinstance(src1, int):

            asm.append(f"mov eax, {src1}")

        else:

            if "dword [" in loc\_src1:

                asm.append(f"mov eax, dword {loc\_src1}")

            else:

                asm.append(f"mov eax, {loc\_src1}")

        # Perform operation

        if isinstance(src2, int):

            asm.append(f"add eax, {src2}")

        else:

            if "dword [" in loc\_src2:

                asm.append(f"add eax, dword {loc\_src2}")

            else:

                asm.append(f"add eax, {loc\_src2}")

        # Store result

        if "dword [" in loc\_dest:

            asm.append(f"mov dword {loc\_dest}, eax")

        else:

            asm.append(f"mov {loc\_dest}, eax")

        return "\n".join(asm)

    elif op == "\*":

        dest, src1, src2 = instr[1], instr[2], instr[3]

        loc\_dest = get\_location(dest)

        loc\_src1 = get\_location(src1)

        loc\_src2 = get\_location(src2)

        asm = []

        # Load first operand

        if isinstance(src1, int):

            asm.append(f"mov eax, {src1}")

        else:

            if "dword [" in loc\_src1:

                asm.append(f"mov eax, dword {loc\_src1}")

            else:

                asm.append(f"mov eax, {loc\_src1}")

        # Perform multiplication

        if isinstance(src2, int):

            asm.append(f"imul eax, {src2}")

        else:

            if "dword [" in loc\_src2:

                asm.append(f"imul eax, dword {loc\_src2}")

            else:

                asm.append(f"imul eax, {loc\_src2}")

        # Store result

        if "dword [" in loc\_dest:

            asm.append(f"mov dword {loc\_dest}, eax")

        else:

            asm.append(f"mov {loc\_dest}, eax")

        return "\n".join(asm)

    elif op == ">":

        dest, src1, src2 = instr[1], instr[2], instr[3]

        loc\_dest = get\_location(dest)

        loc\_src1 = get\_location(src1)

        loc\_src2 = get\_location(src2)

        asm = []

        # Load first operand

        if isinstance(src1, int):

            asm.append(f"mov eax, {src1}")

        else:

            if "dword [" in loc\_src1:

                asm.append(f"mov eax, dword {loc\_src1}")

            else:

                asm.append(f"mov eax, {loc\_src1}")

        # Compare

        if isinstance(src2, int):

            asm.append(f"cmp eax, {src2}")

        else:

            if "dword [" in loc\_src2:

                asm.append(f"cmp eax, dword {loc\_src2}")

            else:

                asm.append(f"cmp eax, {loc\_src2}")

        # Set result

        asm.append("setg al")

        asm.append("movzx eax, al")

        # Store result

        if "dword [" in loc\_dest:

            asm.append(f"mov dword {loc\_dest}, eax")

        else:

            asm.append(f"mov {loc\_dest}, eax")

        return "\n".join(asm)

    elif op == "ifnot":

        cond, label = instr[1], instr[2]

        loc\_cond = get\_location(cond)

        if isinstance(cond, int):

            if cond == 0:

                return f"jmp {label}"

            else:

                return f"; condition is always true"

        if "dword [" in loc\_cond:

            return f"cmp dword {loc\_cond}, 0\nje {label}"

        else:

            return f"cmp {loc\_cond}, 0\nje {label}"

    elif op == "goto":

        return f"jmp {instr[1]}"

    elif op == "label":

        return f"{instr[1]}:"

    elif op == "print":

        expr = instr[1]

        if isinstance(expr, str) and expr.startswith('"'):

            if expr == '"Result is greater than 40"':

                return f"push str1\npush fmt\_str\ncall \_printf\nadd esp, 8"

            else:

                return f"; Unhandled string: {expr}"

        else:

            loc = get\_location(expr)

            if "dword [" in loc:

                return f"mov eax, dword {loc}\npush eax\npush fmt\_int\ncall \_printf\nadd esp, 8"

            else:

                return f"push {loc}\npush fmt\_int\ncall \_printf\nadd esp, 8"

    return f"; unhandled op: {op}"

**output.s:**

global \_main

extern \_printf

extern \_exit

section .data

fmt\_int db "%d", 10, 0

fmt\_str db "%s", 10, 0

msg db "Operation result stored", 10, 0

str1 db "Result is greater than 40", 10, 0

section .bss

res resb 4

section .text

\_main:

push ebp

mov ebp, esp

sub esp, 64

mov eax, 23

mov ebx, 54

mov ecx, 0

mov eax, eax

imul eax, 4

mov edx, eax

mov ecx, edx

mov dword dword [ebp-4], 0

mov dword dword [ebp-8], 0

mov eax, eax

add eax, ebx

mov dword dword [ebp-12], eax

mov eax, dword [ebp-12]

mov dword dword [ebp-16], eax

mov eax, eax

add eax, ebx

mov dword dword [ebp-20], eax

mov eax, dword [ebp-20]

mov dword dword [ebp-24], eax

; unhandled op: <

cmp dword dword [ebp-28], 0

je L1

mov eax, dword dword [ebp-8]

add eax, dword dword [ebp-16]

mov dword dword [ebp-32], eax

mov eax, dword [ebp-32]

mov dword dword [ebp-8], eax

mov eax, dword dword [ebp-4]

add eax, 1

mov dword dword [ebp-36], eax

mov eax, dword [ebp-36]

mov dword dword [ebp-4], eax

jmp L2

L1:

L2:

mov eax, ecx

cmp eax, 40

setg al

movzx eax, al

mov dword dword [ebp-40], eax

cmp dword dword [ebp-40], 0

je L3

push str1

push fmt\_str

call \_printf

add esp, 8

mov eax, dword dword [ebp-24]

push eax

push fmt\_int

call \_printf

add esp, 8

jmp L4

L3:

L4:

; Program exit

push msg

push fmt\_str

call \_printf

add esp, 8

mov esp, ebp

pop ebp

push 0

call \_exit

**Final.py**

import os

import subprocess

from lexer import lexer

from syntax import parser

from semantic import analyze

from Ast\_Table import generate\_ast\_dot

from IR import generate\_ir

from OC import optimize\_ir

from codegen import generate\_assembly

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

    base\_dir = r"C:\Users\abdul\OneDrive\Desktop\CC"

    input\_file = os.path.join(base\_dir, "test\_code.txt")

    ir\_file = os.path.join(base\_dir, "ir.txt")

    optimized\_ir\_file = os.path.join(base\_dir, "optimized\_ir.txt")

    ast\_dot\_file = os.path.join(base\_dir, "ast.dot")

    assembly\_file = os.path.join(base\_dir, "output.s")

    if not os.path.exists(base\_dir):

        print(f"Error: Directory {base\_dir} does not exist.")

        exit(1)

    try:

        print(f"Reading input from {input\_file}")

        with open(input\_file, "r") as file:

            data = file.read()

    except FileNotFoundError as e:

        print(f"Error: Input file not found - {e}")

        exit(1)

    except Exception as e:

        print(f"Unexpected Error while reading input file: {e}")

        exit(1)

    print("\n      Lexical Analysis ")

    try:

        lexer.input(data)

        for tok in lexer:

            print(tok)

    except Exception as e:

        print(f"Lexical Analysis Error: {e}")

        exit(1)

    print("\n      Syntax Analysis ")

    try:

        ast = parser.parse(data, lexer=lexer)

        if ast is None:

            print("Error: Parsing failed, AST is None.")

            exit(1)

        print("Generated AST:", ast)

    except Exception as e:

        print(f"Syntax Analysis Error: {e}")

        exit(1)

    print("\n      Semantic Analysis ")

    semantic\_passed = False

    try:

        analyze(ast)

        semantic\_passed = True

        print("Semantic analysis passed successfully.")

    except Exception as e:

        print(f"Semantic Error: {e}. Continuing to later phases for debugging.")

    print("\n      AST Visualization ")

    try:

        if os.path.exists(ast\_dot\_file):

            print(f"Warning: {ast\_dot\_file} already exists, overwriting.")

        test\_file = os.path.join(base\_dir, "test\_write.txt")

        with open(test\_file, "w") as f:

            f.write("test")

        print(f"Successfully wrote test file to {test\_file}")

        os.remove(test\_file)

        if not generate\_ast\_dot(ast, filename=ast\_dot\_file):

            print("Failed to generate AST visualization, but continuing.")

        else:

            print("AST visualization generated successfully.")

            png\_file = ast\_dot\_file.replace('.dot', '.png')

            subprocess.run(["dot", "-Tpng", ast\_dot\_file, "-o", png\_file], check=True)

            if os.path.exists(png\_file):

                print(f"Confirmed: {png\_file} exists.")

            else:

                print(f"Error: {png\_file} was not created.")

    except subprocess.CalledProcessError as e:

        print(f"Graphviz Error: Ensure 'dot' is installed and in PATH. Error: {e}")

    except Exception as e:

        print(f"AST Visualization Error: {e}. Continuing to IR generation.")

    print("\n      Intermediate Representation (IR) Generation ")

    try:

        ir = generate\_ir(ast)

        print("IR Instructions:")

        with open(ir\_file, "w") as f:

            for instr in ir:

                print(instr)

                f.write(str(instr) + "\n")

        print(f"IR saved to {ir\_file}")

    except Exception as e:

        print(f"IR Generation Error: {e}. Continuing to optimization.")

    print("\n      Optimized IR ")

    try:

        optimized\_ir = optimize\_ir(ir)

        print("Optimized IR Instructions:")

        with open(optimized\_ir\_file, "w") as f:

            for instr in optimized\_ir:

                print(instr)

                f.write(str(instr) + "\n")

        print(f"Optimized IR saved to {optimized\_ir\_file}")

    except Exception as e:

        print(f"IR Optimization Error: {e}. Continuing to code generation.")

    print("\n      Code Generation ")

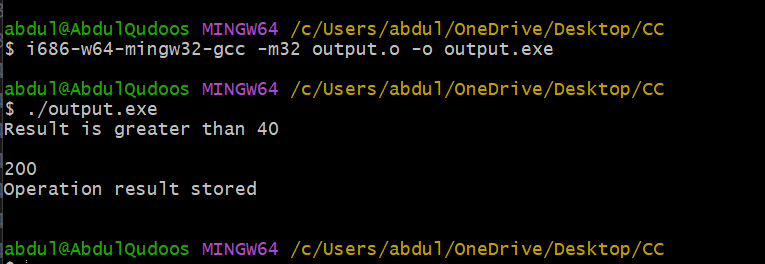
    try:

        generate\_assembly(optimized\_ir, filename=assembly\_file)

        print(f"Assembly code saved to {assembly\_file}")

    except Exception as e:

        print(f"Code Generation Error: {e}")



* nasm -f win32 output.s -o output.o
* i686-w64-mingw32-gcc -m32 output.o -o output.exe
* ./output.exe