University Of Asia Pacific

Department of Computer Science and Engineering

Course Code: CSE 430

Course Title: Compiler Design Lab

Lab Exercise: Compiler Design Lab Project

Submitted To,

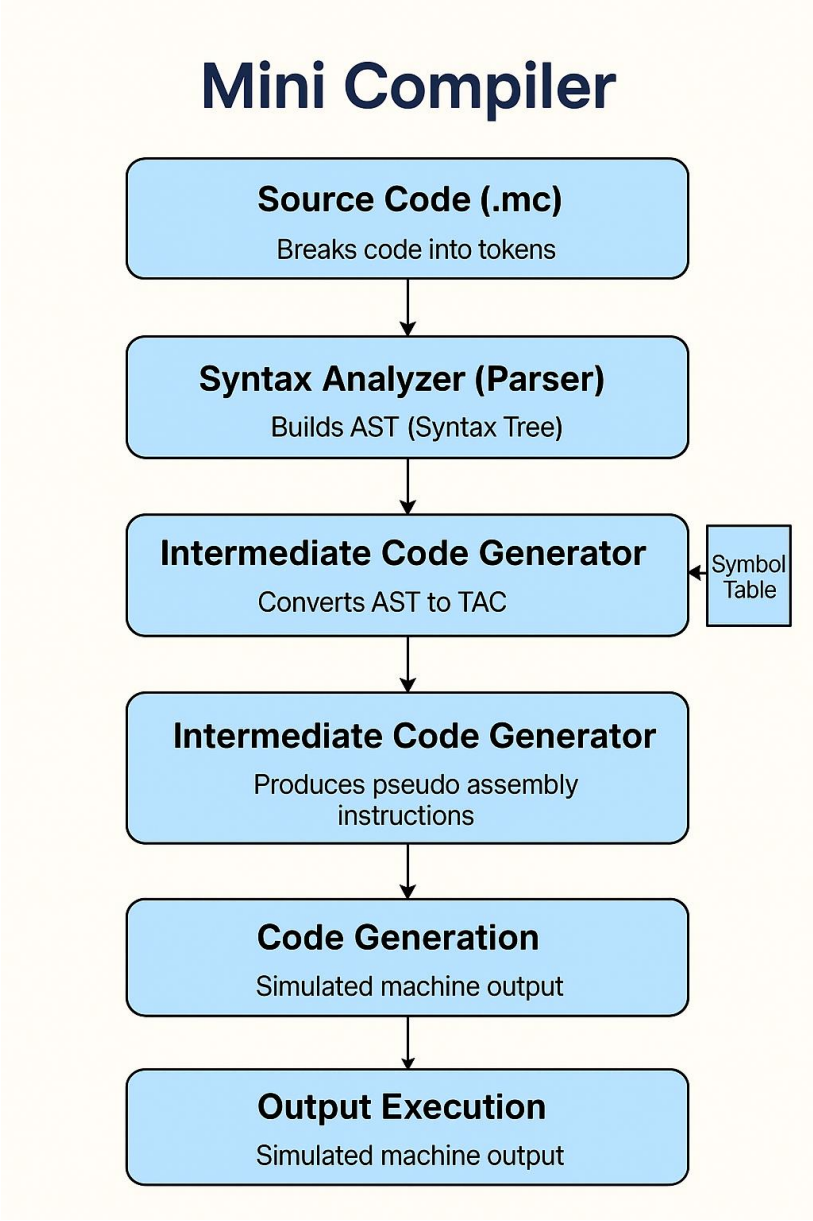
Jayonto Dutta Plabon
Lecturer, Department of CSE
University of Asia Pacific

Submitted By,

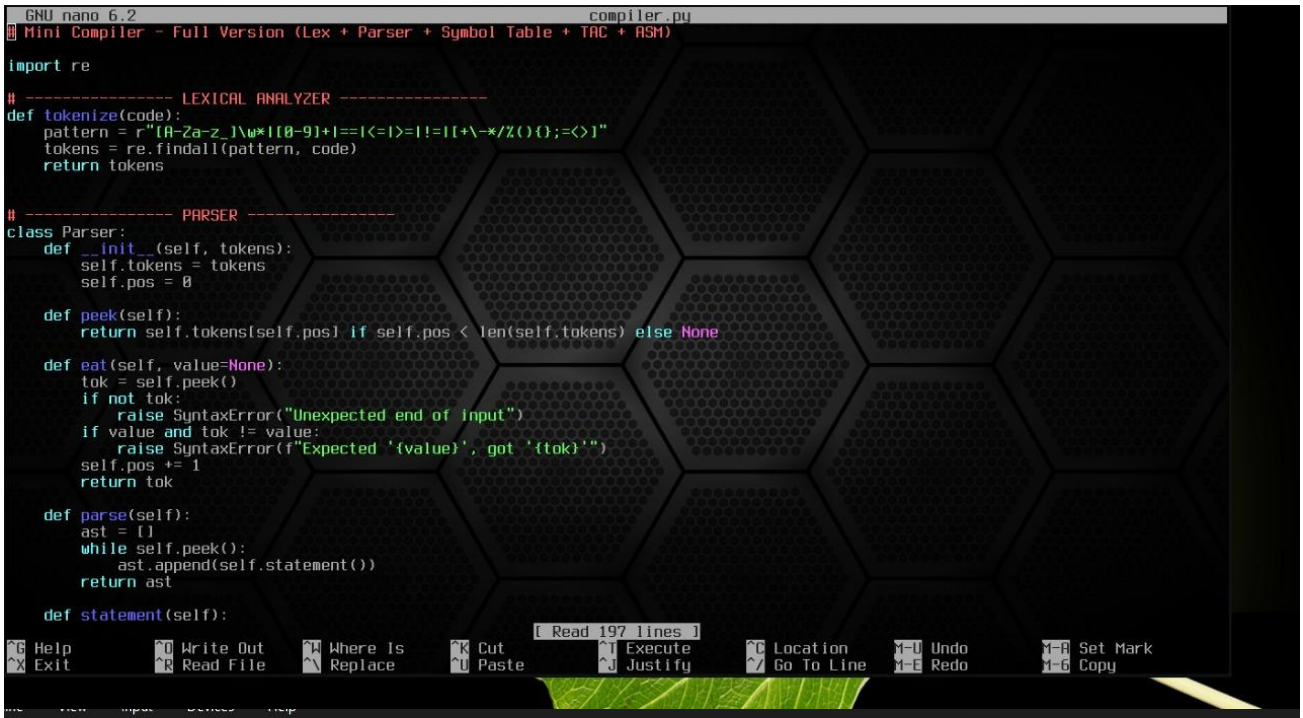
Name: Sadia Islam Priya
ID: 21201181
Section: D2

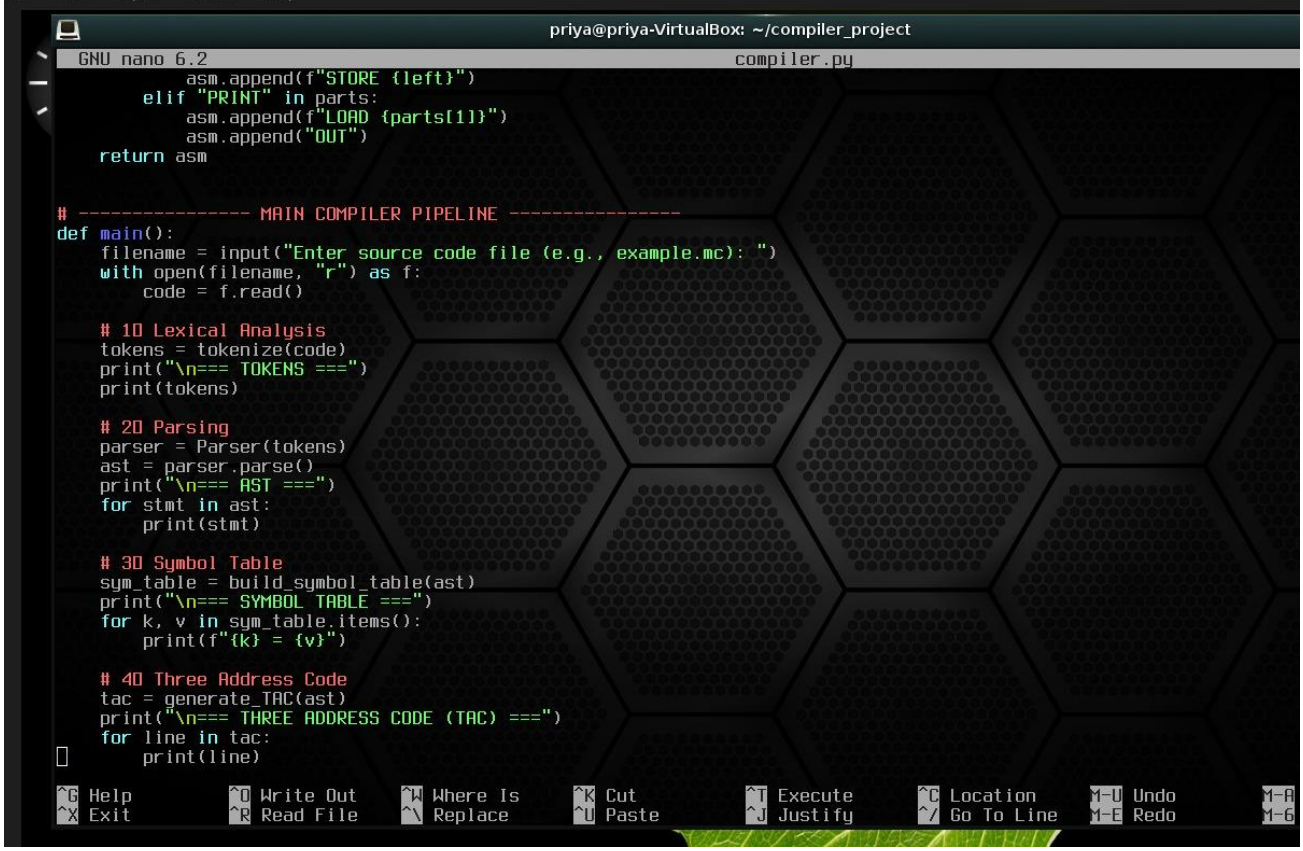
Task: Create a mini compiler that takes a small amount of code as input and it generates the machine code after compilation. So, basically, you need to merge all the phases and apply what you have learned till now to create the compiler. You can use various compiler construction tools and use any programming language, including Lex/ YACC.

Block diagram:

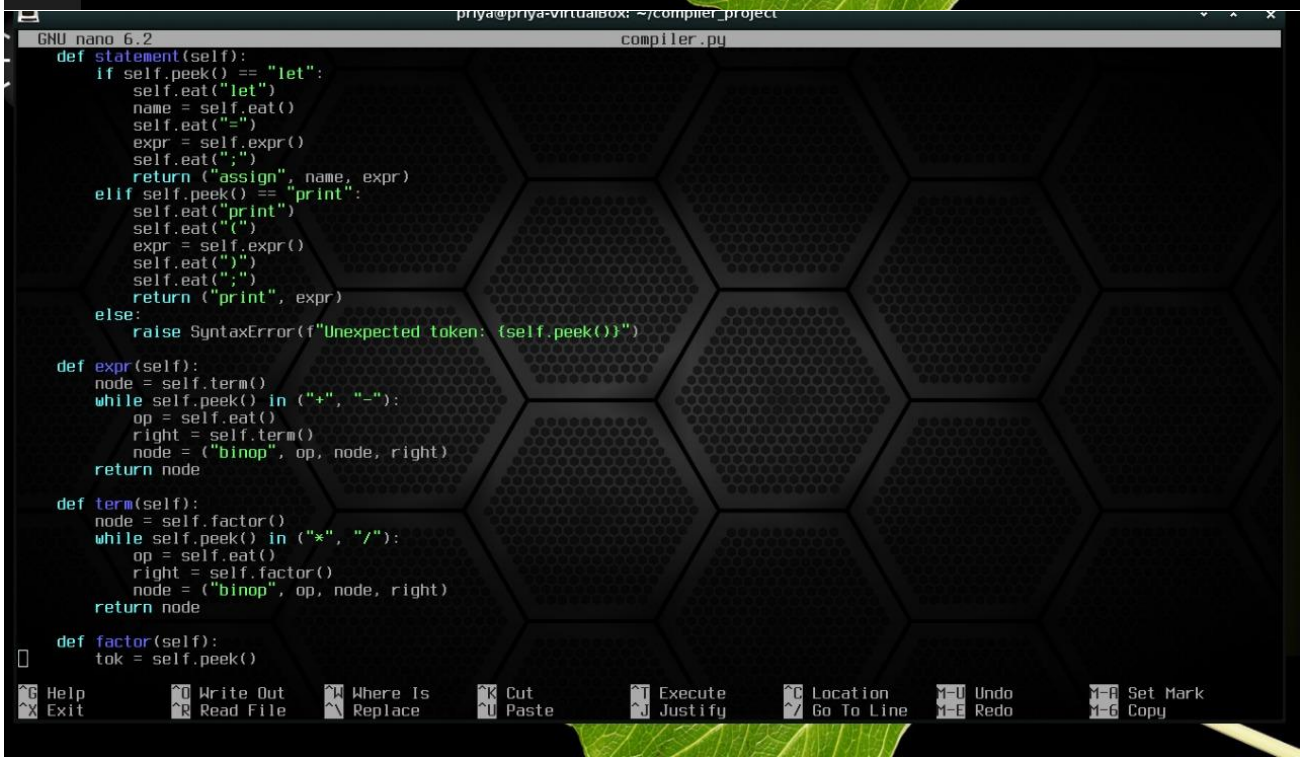


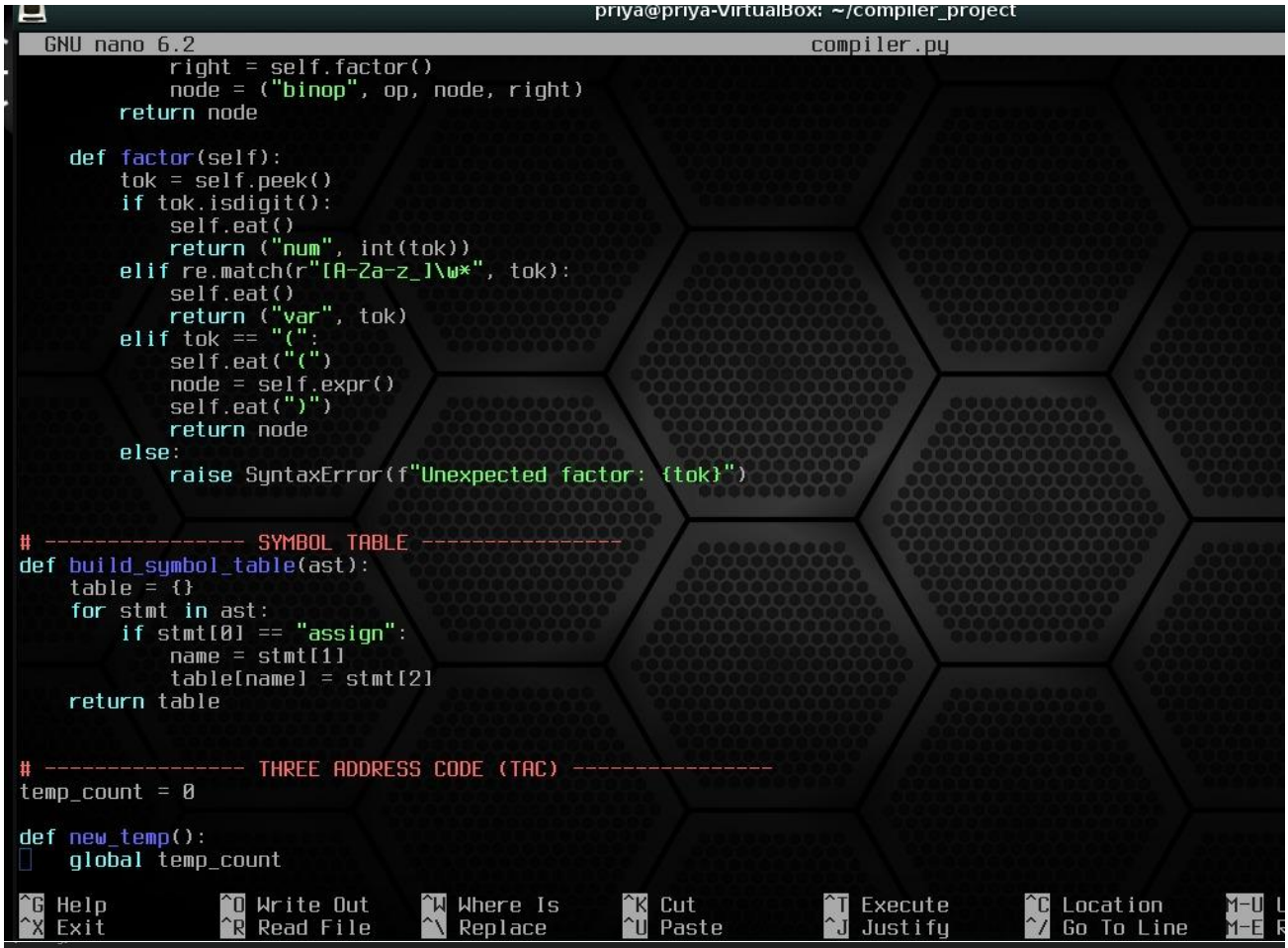
Main code:

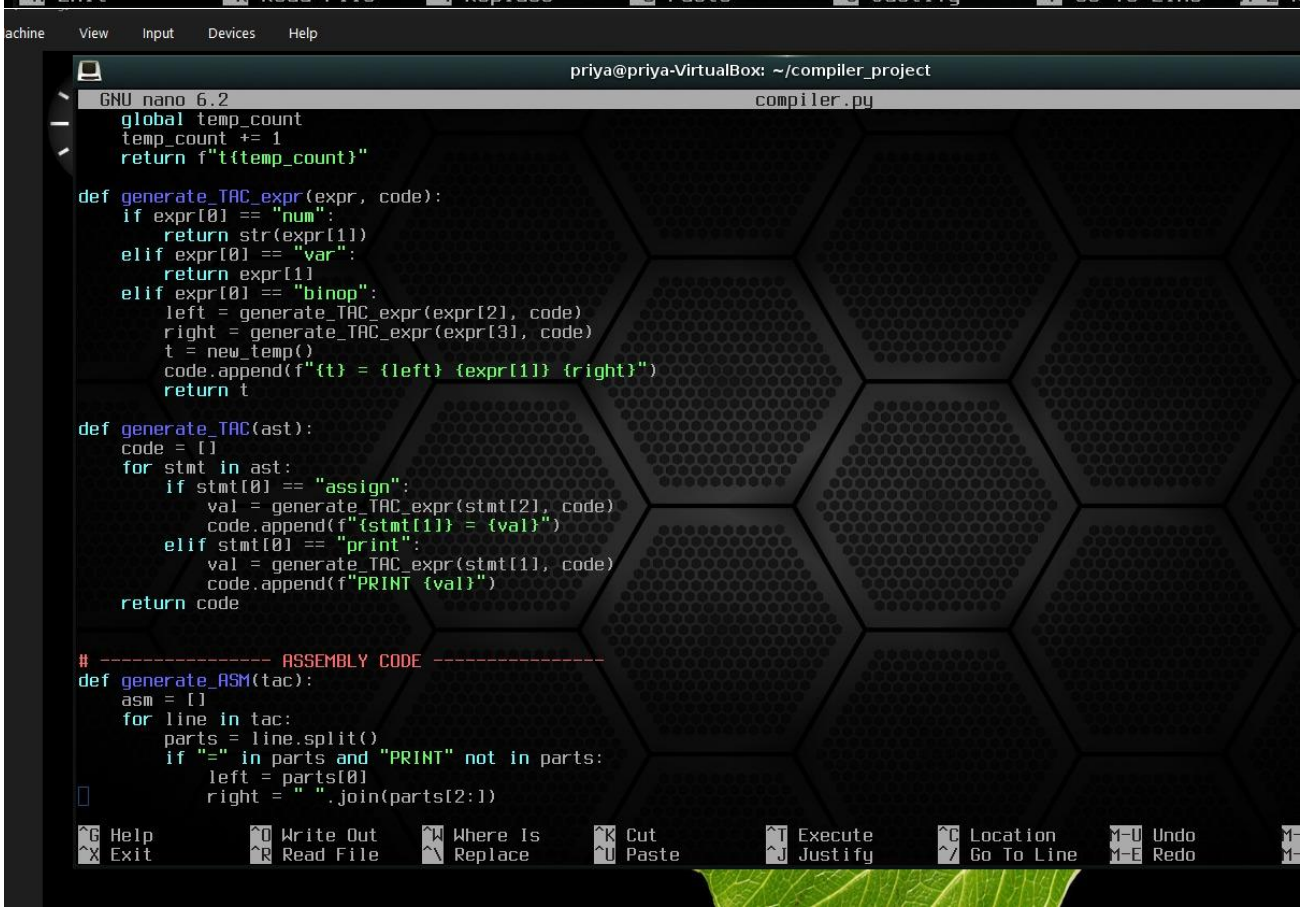

GNU nano 6.2
compiler.py
# Mini Compiler - Full Version (Lex + Parser + Symbol Table + TAC + ASM)
import re
# ---------- LEXICAL ANALYZER ----------
def tokenize(code):
pattern = r"[A-Za-z_]\w*|[0-9]+|==|<=|>=|!=|[+\-*/%(){};=<>]"
tokens = re.findall(pattern, code)
return tokens
# ---------- PARSER ----------
class Parser:
def __init__(self, tokens):
self.tokens = tokens
self.pos = 0
def peek(self):
return self.tokens[self.pos] if self.pos < len(self.tokens) else None
def eat(self, value=None):
tok = self.peek()
if not tok:
raise SyntaxError("Unexpected end of input")
if value and tok != value:
raise SyntaxError(f"Expected '{value}', got '{tok}'")
self.pos += 1
return tok
def parse(self):
ast = []
while self.peek():
ast.append(self.statement())
return ast
def statement(self):
[Read 197 lines]
^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location M-U Undo M-A Set Mark
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line M-E Redo M-6 Copy


priya@priya-VirtualBox: ~/compiler_project
GNU nano 6.2
compiler.py
asm.append(f"STORE {left}")
elif "PRINT" in parts:
asm.append(f"LOAD {parts[1]}")
asm.append("OUT")
return asm
# ---------- MAIN COMPILER PIPELINE ----------
def main():
filename = input("Enter source code file (e.g., example.mc): ")
with open(filename, "r") as f:
code = f.read()
# 1D Lexical Analysis
tokens = tokenize(code)
print("\n=== TOKENS ===")
print(tokens)
# 2D Parsing
parser = Parser(tokens)
ast = parser.parse()
print("\n=== AST ===")
for stmt in ast:
print(stmt)
# 3D Symbol Table
sym_table = build_symbol_table(ast)
print("\n=== SYMBOL TABLE ===")
for k, v in sym_table.items():
print(f"{k} = {v}")
# 4D Three Address Code
tac = generate_TAC(ast)
print("\n=== THREE ADDRESS CODE (TAC) ===")
for line in tac:
print(line)
^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location M-U Undo M-A
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line M-E Redo M-6

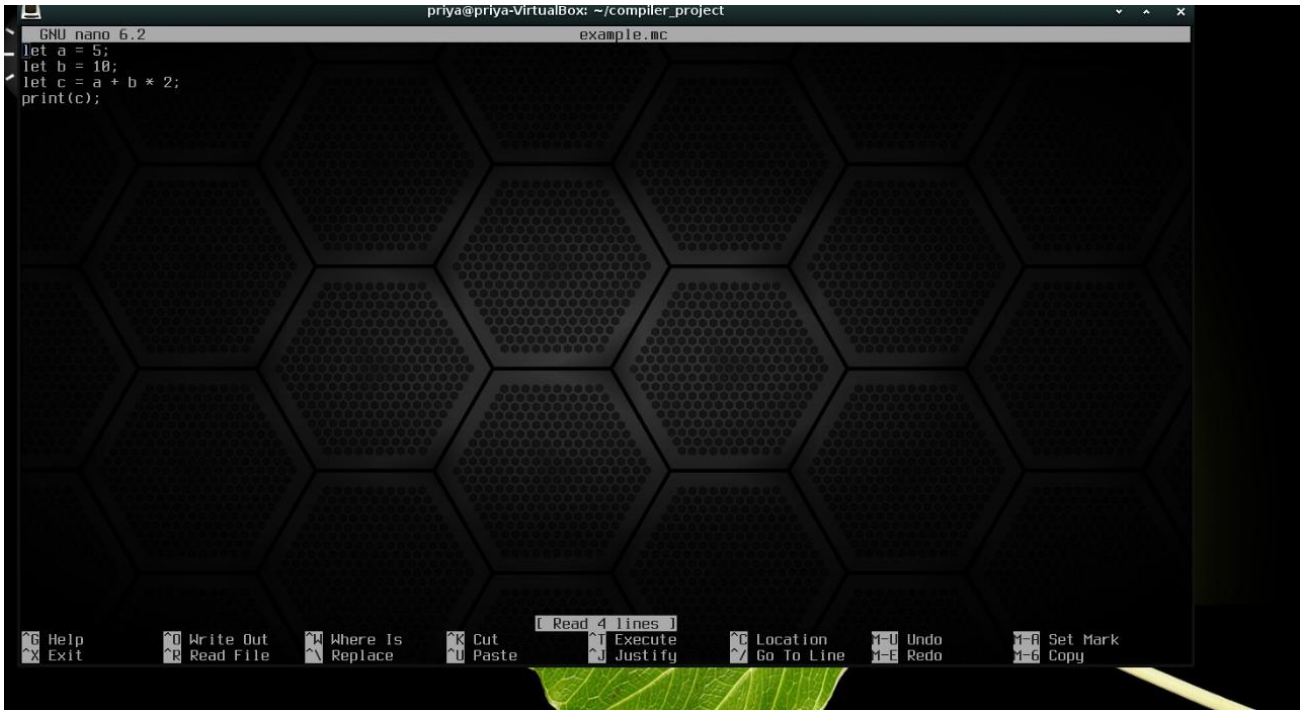

GNU nano 6.2
# 3D Symbol Table
sym_table = build_symbol_table(ast)
print("\n=== SYMBOL TABLE ===")
for k, v in sym_table.items():
print(f"{k} = {v}")
# 4D Three Address Code
tac = generate_TAC(ast)
print("\n=== THREE ADDRESS CODE (TAC) ===")
for line in tac:
print(line)
# 5D Assembly Code
asm = generate_ASM(tac)
print("\n=== ASSEMBLY CODE ===")
for line in asm:
print(line)
# 6D Final Output Simulation
print("\n=== OUTPUT ===")
memory = {}
for stmt in tac:
if "=" in stmt and "PRINT" not in stmt:
var, expr = stmt.split(" = ")
try:
memory[var] = eval(expr, {}, memory)
except:
memory[var] = 0
elif "PRINT" in stmt:
var = stmt.split()[1]
print(memory.get(var, 0))
if __name__ == "__main__":
main()
^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location M-U Undo M-A Set Mark
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line M-E Redo M-6 Copy


priya@priya-VirtualBox: ~/compiler_project
GNU nano 6.2
compiler.py
def statement(self):
if self.peek() == "let":
self.eat("let")
name = self.eat()
self.eat("=")
expr = self.expr()
self.eat(";")
return ("assign", name, expr)
elif self.peek() == "print":
self.eat("print")
self.eat("(")
expr = self.expr()
self.eat(")")
self.eat(";")
return ("print", expr)
else:
raise SyntaxError(f"Unexpected token: {self.peek()}")
def expr(self):
node = self.term()
while self.peek() in ("+", "-"):
op = self.eat()
right = self.term()
node = ("binop", op, node, right)
return node
def term(self):
node = self.factor()
while self.peek() in ("*", "/"):
op = self.eat()
right = self.factor()
node = ("binop", op, node, right)
return node
def factor(self):
tok = self.peek()
^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location M-U Undo M-A Set Mark
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line M-E Redo M-6 Copy

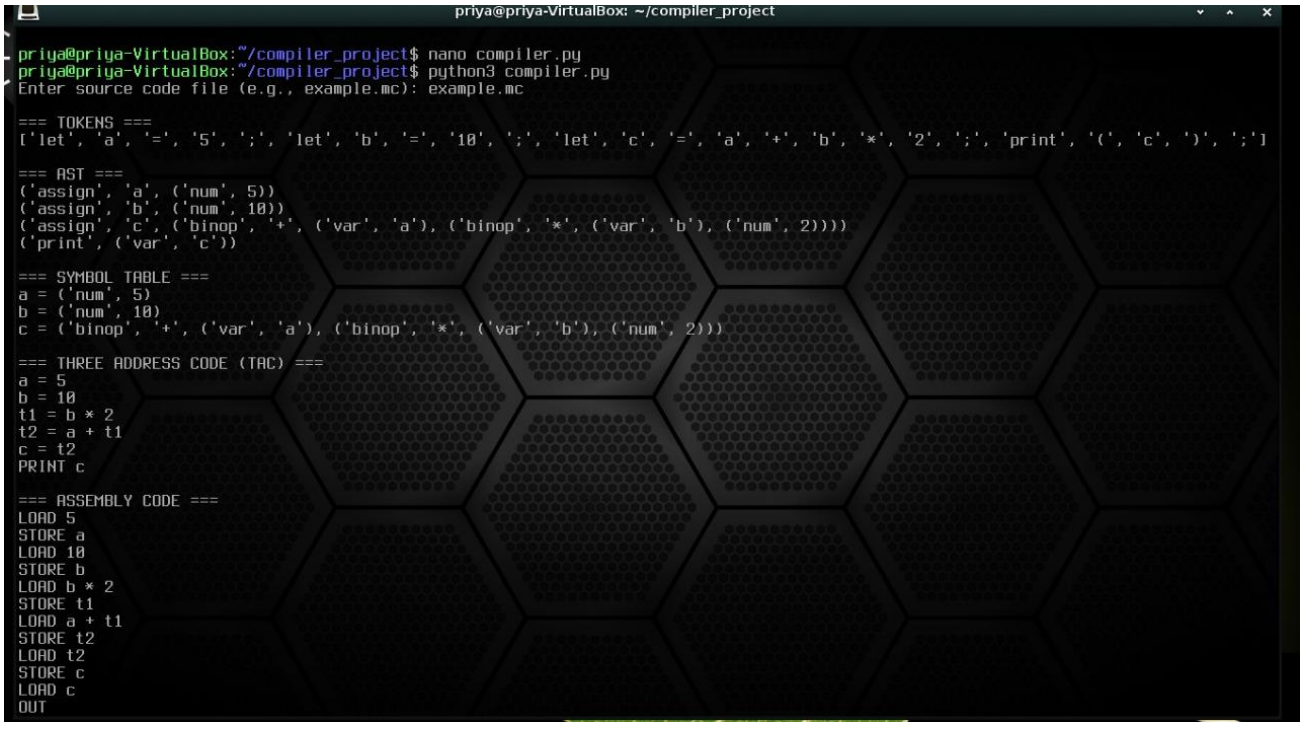

priya@priya-VirtualBox: ~/compiler_project
GNU nano 6.2
compiler.py
right = self.factor()
node = ("binop", op, node, right)
return node
def factor(self):
tok = self.peek()
if tok.isdigit():
self.eat()
return ("num", int(tok))
elif re.match(r"[A-Za-z_]\w*", tok):
self.eat()
return ("var", tok)
elif tok == "(":
self.eat("(")
node = self.expr()
self.eat(")")
return node
else:
raise SyntaxError(f"Unexpected factor: {tok}")
# ---------------- SYMBOL TABLE ----------------
def build_symbol_table(ast):
table = {}
for stmt in ast:
if stmt[0] == "assign":
name = stmt[1]
table[name] = stmt[2]
return table
# ---------------- THREE ADDRESS CODE (TAC) ----------------
temp_count = 0
def new_temp():
global temp_count
^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location M-U
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line M-E

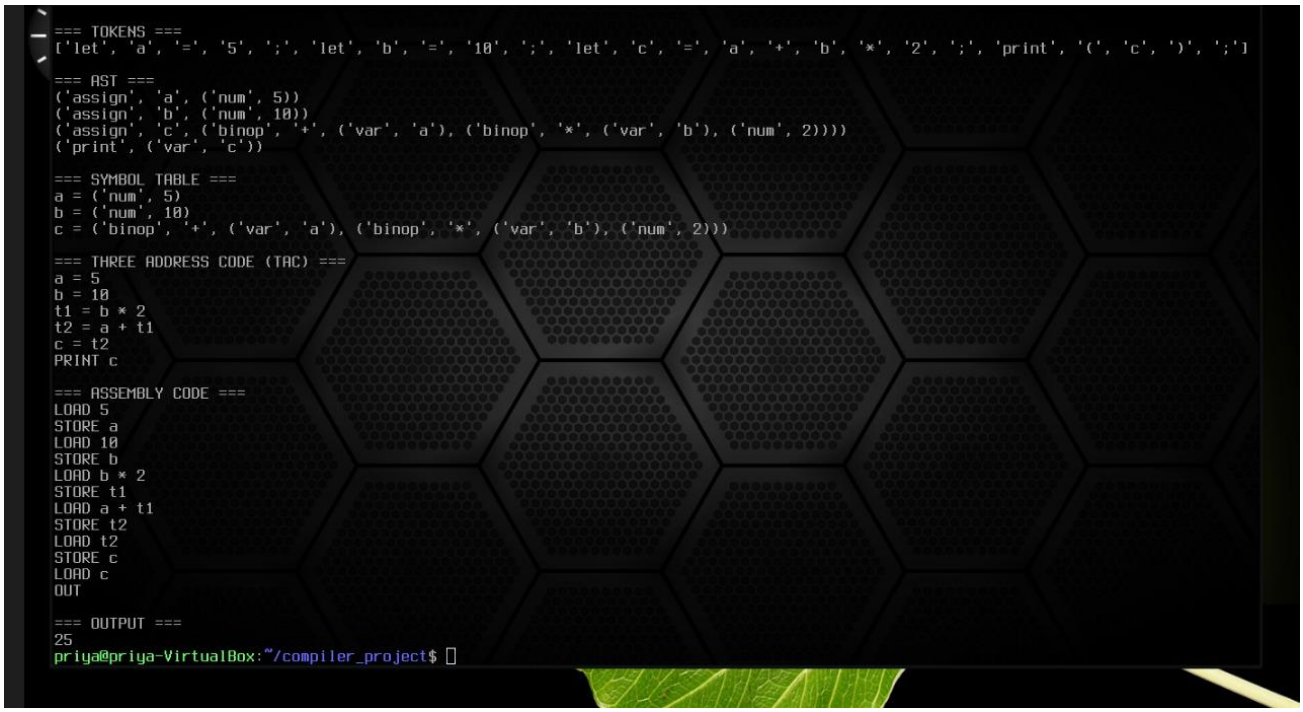

achine View Input Devices Help
priya@priya-VirtualBox: ~/compiler_project
GNU nano 6.2
compiler.py
global temp_count
temp_count += 1
return f"t{temp_count}"
def generate_TAC_expr(expr, code):
if expr[0] == "num":
return str(expr[1])
elif expr[0] == "var":
return expr[1]
elif expr[0] == "binop":
left = generate_TAC_expr(expr[2], code)
right = generate_TAC_expr(expr[3], code)
t = new_temp()
code.append(f"{t} = {left} {expr[1]} {right}")
return t
def generate_TAC(ast):
code = []
for stmt in ast:
if stmt[0] == "assign":
val = generate_TAC_expr(stmt[2], code)
code.append(f"{stmt[1]} = {val}")
elif stmt[0] == "print":
val = generate_TAC_expr(stmt[1], code)
code.append(f"PRINT {val}")
return code
# ---------------- ASSEMBLY CODE ----------------
def generate_ASM(tac):
asm = []
for line in tac:
parts = line.split()
if "=" in parts and "PRINT" not in parts:
left = parts[0]
right = " ".join(parts[2:])
^G Help ^O Write Out ^W Where Is ^K Cut ^T Execute ^C Location M-U Undo
^X Exit ^R Read File ^\ Replace ^U Paste ^J Justify ^/ Go To Line M-E Redo

Input:



Output:





Implementation:

It performs all major compiler phases — Lexical Analysis, Syntax Analysis, Symbol Table Construction, Intermediate Code Generation, Assembly Code Generation, and Output Execution. All phases were implemented manually without using external tools such as Lex.

The compiler takes a simple program written in a C-like syntax and processes it step by step. It first tokenizes the input, builds an Abstract Syntax Tree (AST), stores variables in a Symbol Table, and then generates Three Address Code (TAC), followed by pseudo Assembly Code. Finally, it executes the generated code to produce the output.

Example of the compilation process:

Source Code → Tokens → AST → Symbol Table → TAC → Assembly Code → Output

Example Workflow

Input Source File: example.mc

Compiler Execution Command:

python3 compiler.py