EXPT 1

```
from sys import exit

MOT = {
    "STOP": (0, 1), "ADD": (1, 1), "SUB": (2, 1), "MULT": (3, 1),
    "MOVER": (4, 1), "MOVEM": (5, 1), "COMP": (6, 1), "BC": (7, 1),
    "DIV": (8, 1), "READ": (9, 1), "PRINT": (10, 1), "START": (1, 1),
    "END": (2, 1), "EQU": (3, 1), "ORIGIN": (4, 1), "LTORG": (5, 1),
    "DS": (1, 1), "DC": (2, 1), "AREG": (1, 1), "BREG": (2, 1),
    "CREG": (3, 1), "DREG": (4, 1), "A": (1, 1), "B": (2, 1)
}

l = []
relativeAddress = []
machineCode = []
RA = 0

n = int(input("Enter the no of instruction lines : "))
for i in range(n):
    instructions = input(f"Enter instruction line {i+1}: ").upper()
    l.append(instructions)

for i, x in enumerate(l):
    tokens = x.split()

    if tokens[0] not in MOT:
        print("Instruction is not in Op Code Table.")
        exit(0)

    op_code, size = MOT[tokens[0]]
    RA += size

    if len(tokens) == 1:
        machineCode.append(str(op_code))
    elif tokens[1].isalpha():
        machineCode.append(f"{op_code}")
    else:
        b = tokens[1]
        b = ' '.join([b[j:j+2] for j in range(0, len(b), 2)])
        machineCode.append(f"{op_code} {b}")

    relativeAddress.append(RA - size)

print("Relative Address Instruction OpCode")
for i in range(n):
    print(f"{relativeAddress[i]} {l[i]} {machineCode[i]}")
```

EXPT 4

```
KEYWORDS = ["for", "while", "if", "else", "def", "return", "in", "not",
"and", "or", "print", "range", "input"]
FUNCTIONS = ["len", "int", "str", "float", "bool", "list", "tuple", "dict",
"set", "sorted", "max", "min"]
OPERATORS = ["+", "-", "*", "/", "%", "//", "**", "+=", "-=", "=", "/=",
"==", "!=", "<", ">", "<=", ">=", "not", "in", "and", "or"]

def parse_code(code):
    for line in code:
        parts = line.split(" ")
        for part in parts:
            if part in KEYWORDS:
                print("Keyword: " + part)
            elif part in FUNCTIONS:
                print("Function: " + part)
            elif part in OPERATORS:
                print("Operator: " + part)
            elif part.isnumeric():
                print("Number: " + part)
            else:
                print("Identifier: " + part)

code = [
    "def main():",
    "a = 5",
    "b = 7",
    "if (a > b):",
    "    print('a is greater')",
    "else:",
    "    print('b is greater')",
    "print(a + b)",
    "return 0"
]

parse_code(code)
```

EXPT 5

```
import re

class Token:
    def __init__(self, token_type, value):
        self.token_type = token_type
        self.value = value

class Parser:
    def __init__(self, text):
        self.tokens = self.tokenize(text)
        self.pos = 0

    def parse(self):
        return self.expr()

    def tokenize(self, text):
        token_exprs = [
            (r'\d+', 'INT'),
            (r'\+', 'PLUS'),
            (r'-', 'MINUS'),
            (r'\*', 'MULTIPLY'),
            (r'/', 'DIVIDE'),
            (r'\(', 'LPAREN'),
            (r'\)', 'RPAREN'),
            (r'\s+', None)  # skip whitespace
        ]
        tokens = []
        pos = 0
        while pos < len(text):
            match = None
            for token_expr in token_exprs:
                pattern, token_type = token_expr
                regex = re.compile(pattern)
                match = regex.match(text, pos)
                if match:
                    value = match.group(0)
                    if token_type:
                        token = Token(token_type, value)
                        tokens.append(token)
                    break
            if not match:
                raise ValueError(f'Invalid input at position {pos}')
            else:
                pos = match.end(0)
        return tokens

    def consume(self, token_type):
        if self.pos < len(self.tokens) and self.tokens[self.pos].token_type
== token_type:
            self.pos += 1
        else:
            raise ValueError(f'Expected token type {token_type} at position
{self.pos}')

    def factor(self):
        token = self.tokens[self.pos]
        if token.token_type == 'INT':
```

```
            self.consume('INT')
            return int(token.value)
        elif token.token_type == 'LPAREN':
            self.consume('LPAREN')
            value = self.expr()
            self.consume('RPAREN')
            return value

    def term(self):
        value = self.factor()
        while self.pos < len(self.tokens):
            token = self.tokens[self.pos]
            if token.token_type == 'MULTIPLY':
                self.consume('MULTIPLY')
                value *= self.factor()
            elif token.token_type == 'DIVIDE':
                self.consume('DIVIDE')
                value /= self.factor()
            else:
                break
        return value

    def expr(self):
        value = self.term()
        while self.pos < len(self.tokens):
            token = self.tokens[self.pos]
            if token.token_type == 'PLUS':
                self.consume('PLUS')
                value += self.term()
            elif token.token_type == 'MINUS':
                self.consume('MINUS')
                value -= self.term()
            else:
                break
        return value

text = '2 * (3 + 4) - 5 / 2'
parser = Parser(text)
result = parser.parse()
print(result)  # Output: 12.5
```

EXPT 6

```python
import ast


class IntermediateCodeGenerator(ast.NodeVisitor):
    def __init__(self):
        self.instructions = []
        self.temp_count = 0

    def new_temp(self):
        temp = f"t{self.temp_count}"
        self.temp_count += 1
        return temp

    def visit_Assign(self, node):
        target = node.targets[0].id
        value = self.visit(node.value)
        self.instructions.append(('ASSIGN', target, value))

    def visit_BinOp(self, node):
        left = self.visit(node.left)
        right = self.visit(node.right)
        op = type(node.op).__name__
        temp = self.new_temp()
        self.instructions.append((op, temp, left, right))
        return temp

    def visit_Num(self, node):
        return node.n

    def visit_Name(self, node):
        return node.id

    def visit_Print(self, node):
        value = self.visit(node.values[0])
        self.instructions.append(('PRINT', value))


def generate_intermediate_code(source_code):
    ast_tree = ast.parse(source_code)
    icg = IntermediateCodeGenerator()
    icg.visit(ast_tree)
    return icg.instructions


# Example usage
source_code = """
a = 10
b = 20
c = a + b
print(c)
"""
instructions = generate_intermediate_code(source_code)
for instruction in instructions:
    print(instruction)
```

EXPT 7

```
class CodeGenerator:
    def __init__(self, intermediate_code):
        self.intermediate_code = intermediate_code
        self.generated_code = []

    def generate_code(self):
        self.generated_code.append('_start:')
        for instruction in self.intermediate_code:
            opcode = instruction['opcode']
            operands = instruction['operands']

            if opcode == 'ADD':
                self.add(operands)
            elif opcode == 'SUB':
                self.sub(operands)
            elif opcode == 'MULT':
                self.mult(operands)
            elif opcode == 'DIV':
                self.div(operands)
            else:
                raise ValueError(f"Invalid opcode '{opcode}'")

        self.generated_code.append('MOVER R0, %ebx')
        self.generated_code.append('MOVER R1, %REGA')
        self.generated_code.append('int $0x80')

    def add(self, operands):
        op1, op2, result = operands
        self.generated_code.append(f"MOVER {op1}, %REGA")
        self.generated_code.append(f"ADD {op2}, %REGA")
        self.generated_code.append(f"MOVER %REGA, {result}")

    def sub(self, operands):
        op1, op2, result = operands
        self.generated_code.append(f"MOVER {op1}, %REGA")
        self.generated_code.append(f"SUB {op2}, %REGA")
        self.generated_code.append(f"MOVER %REGA, {result}")

    def mult(self, operands):
        op1, op2, result = operands
        self.generated_code.append(f"MOVER {op1}, %REGA")
        self.generated_code.append(f"MULT {op2}, %REGA")
        self.generated_code.append(f"MOVER %REGA, {result}")

    def div(self, operands):
        op1, op2, result = operands
        self.generated_code.append(f"MOVER {op1}, %eax")
        self.generated_code.append(f"cdq")
        self.generated_code.append(f"DIV {op2}")
        self.generated_code.append(f"MOVER %eax, {result}")


intermediate_code = [
 {'opcode': 'ADD', 'operands': [2, 3, 'result']},
 {'opcode': 'SUB', 'operands': [10, 5, 'result']},
 {'opcode': 'MULT', 'operands': [4, 6, 'result']},
 {'opcode': 'DIV', 'operands': [12, 3, 'result']}
```

```
]
# Generate x86 assembly code
code_generator = CodeGenerator(intermediate_code)
code_generator.generate_code()
assembly_code = '\n'.join(code_generator.generated_code)
# Print generated x86 assembly code
print(assembly_code)
```

EXPT 2