**Code:**

def lexer (input code):

tokens = []

current\_token = ''

for char in input code:

if char.isdigit():

current\_token += char

elif char in ['+', '-', '\*', '/']:

if current\_token:

tokens.append(current\_token)

tokens.append(char)

current\_token = ''

elif char == ' ':

if current\_token:

tokens.append(current\_token)

current\_token = ''

else:

raise Exception(f"Invalid character '{char}' found.")

if current\_token:

tokens.append(current\_token)

return tokens

def parser(tokens):

token\_index = 0

def expression():

nonlocal token\_index

term()

while token\_index < len(tokens) and tokens[token\_index] in ['+', '-']:

token\_index += 1

term()

def term():

nonlocal token\_index

factor()

while token\_index < len(tokens) and tokens[token\_index] in ['\*', '/']:

token\_index += 1

factor()

def factor():

nonlocal token\_index

if token\_index < len(tokens) and tokens[token\_index].isdigit():

token\_index += 1

elif token\_index < len(tokens) and tokens[token\_index] == '(':

token\_index += 1

expression()

if token\_index < len(tokens) and tokens[token\_index] == ')':

token\_index += 1

else:

raise Exception("Missing closing parenthesis.")

else:

raise Exception("Invalid syntax.")

expression()

if token\_index < len(tokens):

raise Exception("Unexpected tokens after expression.")

def generate\_intermediate\_code(tokens):

code = []

temp\_count = 1

def new\_temp():

nonlocal temp\_count

temp\_name = f"t{temp\_count}"

temp\_count += 1

return temp\_name

def generate\_code\_for\_expression():

nonlocal tokens, code

temp = new\_temp()

code.append(f"{temp} = {tokens[0]}")

index = 1

while index < len(tokens):

operator = tokens[index]

operand = tokens[index + 1]

temp\_result = new\_temp()

code.append(f"{temp\_result} = {temp} {operator} {operand}")

temp = temp\_result

index += 2

return temp

result\_temp = generate\_code\_for\_expression()

code.append(f"RESULT = {result\_temp}")

return code

def optimize\_code(intermediate\_code):

# Placeholder for optimization techniques

return intermediate\_code

def generate\_target\_code(intermediate\_code):

target code = []

for line in intermediate\_code:

target\_code.append(line)

return '\n'.join(target code)

**Output:**

Input Code: 3 + (4 \* 5) - 6 / 2

Tokens: ['3', '+', '(', '4', '\*', '5', ')', '-', '6', '/', '2']

Intermediate Code:

t1 = 3

t2 = 4 \* 5

t3 = t1 + t2

t4 = 6 / 2

t5 = t3 - t4

RESULT = t5

Target Code:

t1 = 3

t2 = 4 \* 5

t3 = t1 + t2

t4 = 6 / 2

t5 = t3 - t4

RESULT = t5