WRITE A PROGRAM TO GENERATE ASSEMBLY CODE FROM PREFIX CODE

BY:

MOHAMED HARSHAD M-RA2211003010152

DINESH RAJA M-RA2211003010161

MOHAMMED HANNAD MK- RA2211003010198

CONTENT

OBJECTIVE

- The primary objective of this project is to develop a program that takes expressions in prefix notation as input and produces corresponding assembly language code as output.
- This program aims to demonstrate an understanding of parsing, data transformation, and code generation techniques to convert prefix expressions into executable assembly code.

MODULES

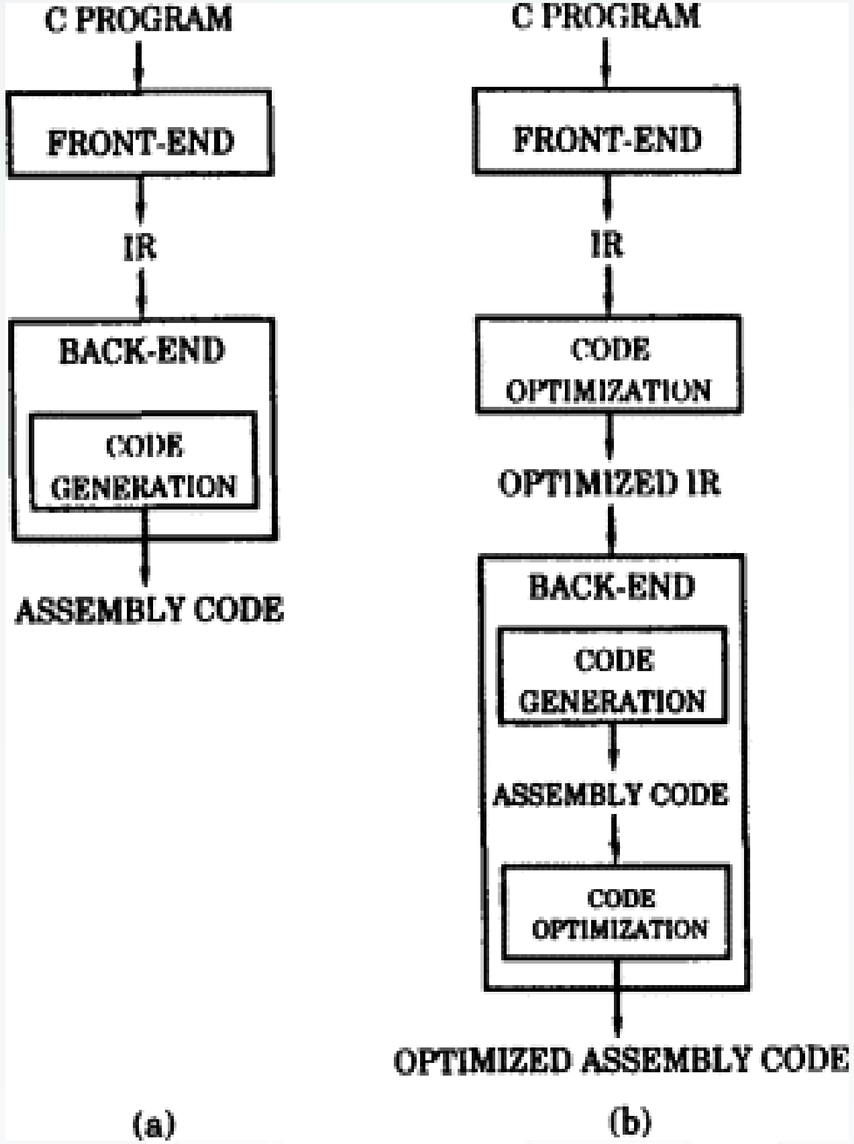
1.Input Processing Module: Responsible for accepting and validating the prefix expression.

2.Parsing and Conversion Module: Converts the prefix expression into a suitable data structure and transforms it into assembly code.

3.Assembly Code Generation Module: Generates assembly code based on the parsed prefix expression.

4.Output Module: Outputs the generated assembly code.

BLOCK DIAGRAM



USES

- **Educational Purposes:**
  - Provides a practical application of data structures and algorithms for parsing and conversion.
  - Demonstrates the link between high-level expressions and low-level assembly instructions, aiding in comprehension of computer architecture.
- **Compiler Design:**
  - Offers a foundation for building a compiler or language interpreter, illustrating the translation of abstract syntax to executable code.

ALGORITHM

Initialize: Start by defining an empty stack, an operator-to-assembly-instruction dictionary (for basic arithmetic operations), and an empty list to store assembly code.

Input: Ask the user to enter a prefix expression. The expression is split into tokens, which are typically numbers and operators (e.g., "+", "-", "*", "/").

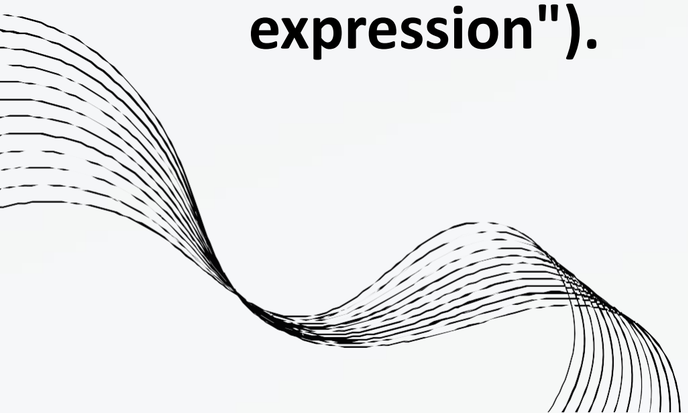
Processing Tokens: Iterate through the tokens in reverse order (right to left), which is a common way to process prefix expressions.

Stack: Use a stack to keep track of operands (numbers) and operators as you process the tokens.

Token Handling:

- If the token is a number, push it onto the stack.
- If the token is an operator (e.g., "+", "-", "*", "/"):
- Pop the top two operands from the stack (the last two numbers added to the stack).
- Use the operator to perform an arithmetic operation on these two operands.
- Push the result back onto the stack.

Error Handling: If the expression is not well-formed or if there's more than one value left on the stack at the end, raise an error (e.g., "Invalid prefix expression").

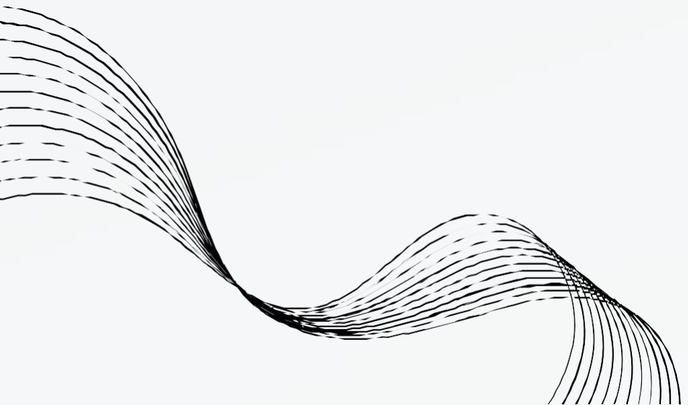


Result: The final result is at the top of the stack.

Generate Assembly Code:

- Move the result to a special register (e.g., "eax" in x86 assembly).
- Optionally, call a function (e.g., "print_result") to print the result. This part is a placeholder, and the actual printing code is not shown.

Output: Return the generated assembly code as a string.



CODE

main.py

```python
def generate_assembly(prefix_expression):
    stack = []

    # Define a dictionary for mapping operators to assembly instructions
    operators = {
        '+': 'add',
        '-': 'sub',
        '*': 'imul',
        '/': 'idiv',
    }

    assembly_code = []

    # Iterate through the tokens in reverse order (right to left)
    for token in reversed(prefix_expression):
        if token.isdigit():
            stack.append(token)  # Push operands onto the stack
        elif token in operators:
            # Pop two operands from the stack
            operand1 = stack.pop()
            operand2 = stack.pop()

            # Generate the corresponding assembly instruction
            instruction = operators[token]

            # Emit the assembly instruction
            assembly_code.append(f'{instruction} {operand1}, {operand2}')
```

```
28
29              # Push the result back onto the stack
30              stack.append(operand1)
31          else:
32              raise ValueError(f"Invalid token: {token}")
33
34      # The final result should be on top of the stack
35      if len(stack) != 1:
36          raise ValueError("Invalid prefix expression")
37
38      # Add code to print the result (assuming x86 assembly)
39      result = stack[0]
40      assembly_code.append(f'mov eax, {result}')
41      assembly_code.append('call print_result')
42
43      return '\n'.join(assembly_code)
44
45  # Get the prefix expression from the user
46  prefix_expression = input("Enter a prefix expression: ").split()
47
48  # Example usage:
49  assembly_code = generate_assembly(prefix_expression)
50  print(assembly_code)
```

CODE LINK:
https://docs.google.com/document/d/11U4r9Vjc3xfo6IB2jslAxFkKqCSKP75eHggMa_P24n8/edit?usp=sharing

OUTPUT

```
Enter a prefix expression: + 5 * 3 4
imul 3, 4
add 5, 3
mov eax, 5
call print_result


...Program finished with exit code 0
Press ENTER to exit console.
```

THANK YOU!