In data structures, \*\*expression evaluation\*\* is the process of evaluating arithmetic expressions represented in infix, postfix, or prefix notation. The goal is to compute the result of an expression, often by using a stack data structure. Expression evaluation is widely used in compilers, interpreters, and calculators.

### Types of Expressions

1. \*\*Infix Expression\*\*: Operands and operators are arranged in the usual human-readable format (e.g., `A + B \* C`).

2. \*\*Prefix Expression (Polish Notation)\*\*: Operators appear before their operands (e.g., `+ A \* B C`).

3. \*\*Postfix Expression (Reverse Polish Notation)\*\*: Operators appear after their operands (e.g., `A B C \* +`).

Each notation has specific rules and use cases.

### Scenarios and Methods for Solving Expression Evaluation

#### 1. \*\*Infix Expression Evaluation\*\*

- In infix expressions, operators are placed between operands. To evaluate, we need to consider \*\*operator precedence\*\* and \*\*associativity\*\*.

- \*\*Scenario\*\*: `3 + 5 \* 2 - 4`

\*\*Approach\*\*:

- Use two stacks: one for operators and one for operands.

- Scan the expression from left to right:

- Push numbers (operands) to the operand stack.

- Push operators to the operator stack after checking precedence and associativity.

- When an operator with lower precedence appears, pop from both stacks, evaluate the result, and push it back to the operand stack.

- At the end, evaluate remaining operations in the stacks to get the final result.

#### 2. \*\*Prefix Expression Evaluation\*\*

- In prefix notation, operators precede their operands. This format eliminates the need for parentheses.

- \*\*Scenario\*\*: `+ 9 \* 2 6`

\*\*Approach\*\*:

- Use a stack to evaluate prefix expressions by scanning the expression from right to left.

- Push operands onto the stack.

- When encountering an operator, pop the required number of operands, apply the operation, and push the result back onto the stack.

- Continue until all elements are processed. The final result will be the only element left in the stack.

#### 3. \*\*Postfix Expression Evaluation\*\*

- In postfix notation, operators come after their operands. Postfix expressions don’t require parentheses or precedence rules.

- \*\*Scenario\*\*: `3 4 + 5 \*`

\*\*Approach\*\*:

- Use a stack to evaluate postfix expressions by scanning from left to right.

- Push operands onto the stack.

- When an operator is encountered, pop the required operands from the stack, apply the operation, and push the result back onto the stack.

- Repeat until all elements are processed. The final result will be at the top of the stack.

#### 4. \*\*Handling Nested Expressions (Infix with Parentheses)\*\*

- Expressions can contain nested operations or parentheses, which changes the order of evaluation.

- \*\*Scenario\*\*: `(3 + 5) \* (2 - 4)`

\*\*Approach\*\*:

- Modify the infix evaluation to handle parentheses by treating each open parenthesis as an operator with the highest precedence.

- When encountering a closing parenthesis, perform operations until the corresponding open parenthesis is reached.

- This ensures the expression within the parentheses is evaluated first, then the rest of the expression.

### Example Code for Postfix Evaluation

Here’s a Python example for evaluating a postfix expression using a stack:

```python

def evaluate\_postfix(expression):

stack = []

for char in expression.split():

if char.isdigit():

stack.append(int(char))

else:

operand2 = stack.pop()

operand1 = stack.pop()

if char == '+':

stack.append(operand1 + operand2)

elif char == '-':

stack.append(operand1 - operand2)

elif char == '\*':

stack.append(operand1 \* operand2)

elif char == '/':

stack.append(operand1 / operand2)

return stack.pop()

# Example usage

expression = "3 4 + 5 \*"

print(evaluate\_postfix(expression)) # Output: 35

```

### Summary

1. \*\*Infix Evaluation\*\*: Use two stacks for operators and operands, handling precedence and associativity.

2. \*\*Prefix Evaluation\*\*: Use a stack, evaluate from right to left.

3. \*\*Postfix Evaluation\*\*: Use a stack, evaluate from left to right.

4. \*\*Nested Expressions\*\*: Handle parentheses with the infix method by adjusting operator precedence.

Each method allows for the structured evaluation of expressions, making stacks particularly powerful for this purpose.

Let’s go through each case and scenario for expression evaluation, describing the algorithm and providing code examples where applicable. We'll cover \*\*infix\*\*, \*\*prefix\*\*, and \*\*postfix\*\* expressions, along with \*\*nested infix expressions\*\*.

### 1. Infix Expression Evaluation

In infix notation, operators are written between operands (e.g., `A + B`). Evaluating infix expressions requires handling \*\*operator precedence\*\* and \*\*associativity\*\* rules.

#### Scenario

Consider the expression:

`3 + 5 \* 2 - 4`

#### Approach

1. \*\*Use Two Stacks\*\*: One for operators (`operator\_stack`) and another for operands (`operand\_stack`).

2. \*\*Scan Left to Right\*\*:

- \*\*Operands\*\*: Push directly to the operand stack.

- \*\*Operators\*\*: Check precedence and associativity. Pop operators from the operator stack when they have higher or equal precedence, then push the current operator.

3. \*\*Handle Remaining Operators\*\*: After scanning, apply any remaining operators in the stack.

#### Example Code

Here’s Python code for evaluating an infix expression:

```python

def precedence(op):

if op in ('+', '-'):

return 1

if op in ('\*', '/'):

return 2

return 0

def apply\_op(a, b, op):

if op == '+': return a + b

if op == '-': return a - b

if op == '\*': return a \* b

if op == '/': return a / b

def evaluate\_infix(expression):

operand\_stack = []

operator\_stack = []

i = 0

while i < len(expression):

if expression[i] == ' ':

i += 1

continue

elif expression[i].isdigit():

val = 0

while i < len(expression) and expression[i].isdigit():

val = (val \* 10) + int(expression[i])

i += 1

operand\_stack.append(val)

i -= 1

elif expression[i] in "+-\*/":

while (operator\_stack and precedence(operator\_stack[-1]) >= precedence(expression[i])):

b = operand\_stack.pop()

a = operand\_stack.pop()

op = operator\_stack.pop()

operand\_stack.append(apply\_op(a, b, op))

operator\_stack.append(expression[i])

i += 1

while operator\_stack:

b = operand\_stack.pop()

a = operand\_stack.pop()

op = operator\_stack.pop()

operand\_stack.append(apply\_op(a, b, op))

return operand\_stack[-1]

# Example usage

expression = "3 + 5 \* 2 - 4"

print(evaluate\_infix(expression)) # Output: 9

```

### 2. Prefix Expression Evaluation

In prefix notation, operators appear before their operands (e.g., `+ 3 \* 5 2`). This eliminates the need for parentheses.

#### Scenario

Consider the prefix expression:

`- + 3 \* 5 2 4`

#### Approach

1. \*\*Use a Stack\*\*: Evaluate the expression from \*\*right to left\*\*.

2. \*\*Operands\*\*: Push them directly to the stack.

3. \*\*Operators\*\*: Pop the required number of operands from the stack, apply the operator, and push the result back onto the stack.

#### Example Code

Here’s Python code for evaluating a prefix expression:

```python

def evaluate\_prefix(expression):

stack = []

# Start from the end of the expression

for i in range(len(expression) - 1, -1, -1):

if expression[i].isdigit():

stack.append(int(expression[i]))

else:

operand1 = stack.pop()

operand2 = stack.pop()

if expression[i] == '+':

stack.append(operand1 + operand2)

elif expression[i] == '-':

stack.append(operand1 - operand2)

elif expression[i] == '\*':

stack.append(operand1 \* operand2)

elif expression[i] == '/':

stack.append(operand1 / operand2)

return stack[0]

# Example usage

expression = "- + 3 \* 5 2 4".split()

print(evaluate\_prefix(expression)) # Output: 9

```

### 3. Postfix Expression Evaluation

In postfix notation, operators appear after their operands (e.g., `3 5 2 \* + 4 -`). Like prefix, postfix also doesn’t require parentheses.

#### Scenario

Consider the postfix expression:

`3 5 2 \* + 4 -`

#### Approach

1. \*\*Use a Stack\*\*: Evaluate the expression from \*\*left to right\*\*.

2. \*\*Operands\*\*: Push them directly to the stack.

3. \*\*Operators\*\*: Pop the required number of operands from the stack, apply the operator, and push the result back onto the stack.

#### Example Code

Here’s Python code for evaluating a postfix expression:

```python

def evaluate\_postfix(expression):

stack = []

for char in expression.split():

if char.isdigit():

stack.append(int(char))

else:

operand2 = stack.pop()

operand1 = stack.pop()

if char == '+':

stack.append(operand1 + operand2)

elif char == '-':

stack.append(operand1 - operand2)

elif char == '\*':

stack.append(operand1 \* operand2)

elif char == '/':

stack.append(operand1 / operand2)

return stack.pop()

# Example usage

expression = "3 5 2 \* + 4 -"

print(evaluate\_postfix(expression)) # Output: 9

```

### 4. Infix Expression with Parentheses (Nested Expression)

Parentheses affect the order of evaluation in infix expressions. They indicate that the expression within them should be evaluated first.

#### Scenario

Consider the expression:

`(3 + 5) \* (2 - 4)`

#### Approach

1. \*\*Handle Parentheses\*\*: Treat `(` as an operator with the highest precedence.

2. \*\*Push and Pop on Encountering Parentheses\*\*:

- When `(` is encountered, push it onto the operator stack.

- When `)` is encountered, pop operators until reaching `(`.

3. \*\*Evaluate Within Parentheses\*\*: Process the subexpression within parentheses before moving forward.

#### Example Code

Here’s Python code for evaluating an infix expression with parentheses:

```python

def evaluate\_infix\_with\_parentheses(expression):

def precedence(op):

return 1 if op in ('+', '-') else 2 if op in ('\*', '/') else 0

def apply\_op(a, b, op):

return a + b if op == '+' else a - b if op == '-' else a \* b if op == '\*' else a / b

operand\_stack = []

operator\_stack = []

i = 0

while i < len(expression):

if expression[i] == ' ':

i += 1

continue

elif expression[i] == '(':

operator\_stack.append(expression[i])

elif expression[i].isdigit():

val = 0

while i < len(expression) and expression[i].isdigit():

val = (val \* 10) + int(expression[i])

i += 1

operand\_stack.append(val)

i -= 1

elif expression[i] == ')':

while operator\_stack and operator\_stack[-1] != '(':

op = operator\_stack.pop()

b = operand\_stack.pop()

a = operand\_stack.pop()

operand\_stack.append(apply\_op(a, b, op))

operator\_stack.pop() # pop the '('

else:

while operator\_stack and precedence(operator\_stack[-1]) >= precedence(expression[i]):

op = operator\_stack.pop()

b = operand\_stack.pop()

a = operand\_stack.pop()

operand\_stack.append(apply\_op(a, b, op))

operator\_stack.append(expression[i])

i += 1

while operator\_stack:

op = operator\_stack.pop()

b = operand\_stack.pop()

a = operand\_stack.pop()

operand\_stack.append(apply\_op(a, b, op))

return operand\_stack[-1]

# Example usage

expression = "(3 + 5) \* (2 - 4)"

print(evaluate\_infix\_with\_parentheses(expression)) # Output: -16

```

### Summary

- \*\*Infix Evaluation\*\*: Use two stacks, handle precedence and associativity.

- \*\*Prefix Evaluation\*\*: Scan right-to-left, using a stack.

- \*\*Postfix Evaluation\*\*: Scan left-to-right, using a stack.

- \*\*Infix with Parentheses\*\*: Use parentheses for controlling precedence in infix expressions, adjust precedence handling for nested parts.

Each method uses the stack efficiently to manage operations and operand order for accurate results.