In [1]:

```
from dsLecture3 import ArrayStack

class Arithmatic:
    def __init__(self):
        self._valStk = None
        self._opStk  = None

    def _prec(self, operator):  # Global function
        operators  = ['+', '-', '*', '/', '<=', '==', '>=', '<', '>', '$']
        precedence = [ 2 ,  2 ,  3 ,  3 ,  1  ,   1 ,   1 ,  1 ,  1 ,  0 ]

        return precedence[operators.index(operator)]

    def _doOp(self):
        x = self._valStk.pop()
        y = self._valStk.pop()
        op = self._opStk.pop()

        self._valStk.push(eval(f'{y}{op}{x}'))

    def _repeatOps(self, refOp):
        while len(self._valStk) > 1 and self._prec(refOp) <= self._prec(self._opStk.top()):
            self._doOp()

    def evaluate(self, expression):
        self._valStk = ArrayStack()
        self._opStk  = ArrayStack()

        for token in expression:
            if type(token) in [int, float]:
                self._valStk.push(token)
            else:
                self._repeatOps(token)
                self._opStk.push(token)

        self._repeatOps('$')

        return self._valStk.top()
```

In [2]:

```
A = Arithmatic()
```

In [3]:

```
A.evaluate([1,'+',3])
```

Out[3]:

```
4
```

In [4]:

```
A.evaluate([14,'<=',4,'-',3,'*',2,'+',7])
```

Out[4]:

```
False
```

In [5]:

```
class Arithmatic_explain:
    def __init__(self):
        self._valStk = None
        self._opStk  = None
```

```
    def _prec(self, operator):  # Global function
        operators  = ['+', '-', '*', '/', '<=', '==', '>=', '<', '>', '$']
        precedence = [ 2 ,  2 ,  3 ,  3 ,  1  ,   1 ,   1 ,  1 ,  1 ,  0 ]

        return precedence[operators.index(operator)]

    def _doOp(self):
        print('    # VAL ', end=''); self._valStk.display_complex()
        x = self._valStk.pop()
        y = self._valStk.pop()
        op = self._opStk.pop()

        print(f'    doOp: Pop {x}, then {y} from valStk. Pop {op} from opStk.')
        print(f'    doOp: Calculate {y}{op}{x}, and put it back to valStk.')

        self._valStk.push(eval(f'{y}{op}{x}'))
        print('    # VAL ', end=''); self._valStk.display_complex()
        print('    #  OP ', end=''); self._opStk.display_complex()


    def _repeatOps(self, refOp):
        prec_ref = self._prec(refOp)
        print(f'  repeatOps: Consume all Ops in opStack, which are higher or equal than {refOp}')
        print('  # VAL ', end=''); self._valStk.display_complex()
        print('  #  OP ', end=''); self._opStk.display_complex()

        while len(self._valStk) > 1 and prec_ref <= self._prec(self._opStk.top()):
            prec_top = self._prec(self._opStk.top())
            print(f'  repeatOps: {refOp} <= {self._opStk.top()}')

            self._doOp()
        if len(self._valStk) <= 1:
            print(f'  repeatOps: Because there is only one value in valStk, it ends here.')
        else:
            print(f'  repeatOps: Because {refOp} > {self._opStk.top()}, it ends here')

    def evaluate(self, expression):
        print('evaluate: Initialize valStk and opStk')
        self._valStk = ArrayStack()
        self._opStk  = ArrayStack()
        print('# VAL ', end=''); self._valStk.display_complex()
        print('#  OP ', end=''); self._opStk.display_complex()


        for token in expression:
            if type(token) in [int, float]:
                print(f'evaluate: {token} is a number. push it to valStk.')
                self._valStk.push(token)
            else:
                print(f'evaluate: {token} is an operator. Call repeatOps.')
                self._repeatOps(token)
                print(f'evalaute: push {token} into opStk')
                self._opStk.push(token)

            print('# VAL ', end=''); self._valStk.display_complex()
            print('#  OP ', end=''); self._opStk.display_complex()

        print(f'evalaute: End of the expression. Consume all the remaining operations')
        self._repeatOps('$')

        return self._valStk.top()
```

In [6]:

```
A_exp = Arithmatic_explain()
```

In [7]:

```
A_exp.evaluate([1,'+',3])
```

```
evaluate: Initialize valStk and opStk
# VAL STACK: B||T
#  OP STACK: B||T
evaluate: 1 is a number. push it to valStk.
```

```
# VAL STACK: B|1|T
#  OP STACK: B||T
evaluate: + is an operator. Call repeatOps.
  repeatOps: Consume all Ops in opStack, which are higher or equal than +
  # VAL STACK: B|1|T
  #  OP STACK: B||T
  repeatOps: Because there is only one value in valStk, it ends here.
evalaute: push + into opStk
# VAL STACK: B|1|T
#  OP STACK: B|+|T
evaluate: 3 is a number. push it to valStk.
# VAL STACK: B|1,3|T
#  OP STACK: B|+|T
evalaute: End of the expression. Consume all the remaining operations
  repeatOps: Consume all Ops in opStack, which are higher or equal than $
  # VAL STACK: B|1,3|T
  #  OP STACK: B|+|T
  repeatOps: $ <= +
    # VAL STACK: B|1,3|T
    doOp: Pop 3, then 1 from valStk. Pop + from opStk.
    doOp: Calculate 1+3, and put it back to valStk.
    # VAL STACK: B|4|T
    #  OP STACK: B||T
  repeatOps: Because there is only one value in valStk, it ends here.
```

Out[7]:

```
4
```

In [8]:

```
A_exp.evaluate([14,'<=',4,'-',3,'*',2,'+',7])
```

```
evaluate: Initialize valStk and opStk
# VAL STACK: B||T
#  OP STACK: B||T
evaluate: 14 is a number. push it to valStk.
# VAL STACK: B|14|T
#  OP STACK: B||T
evaluate: <= is an operator. Call repeatOps.
  repeatOps: Consume all Ops in opStack, which are higher or equal than <=
  # VAL STACK: B|14|T
  #  OP STACK: B||T
  repeatOps: Because there is only one value in valStk, it ends here.
evalaute: push <= into opStk
# VAL STACK: B|14|T
#  OP STACK: B|<=|T
evaluate: 4 is a number. push it to valStk.
# VAL STACK: B|14,4|T
#  OP STACK: B|<=|T
evaluate: - is an operator. Call repeatOps.
  repeatOps: Consume all Ops in opStack, which are higher or equal than -
  # VAL STACK: B|14,4|T
  #  OP STACK: B|<=|T
  repeatOps: Because - > <=, it ends here
evalaute: push - into opStk
# VAL STACK: B|14,4|T
#  OP STACK: B|<=,-|T
evaluate: 3 is a number. push it to valStk.
# VAL STACK: B|14,4,3|T
#  OP STACK: B|<=,-|T
evaluate: * is an operator. Call repeatOps.
  repeatOps: Consume all Ops in opStack, which are higher or equal than *
  # VAL STACK: B|14,4,3|T
  #  OP STACK: B|<=,-|T
  repeatOps: Because * > -, it ends here
evalaute: push * into opStk
# VAL STACK: B|14,4,3|T
#  OP STACK: B|<=,-,*|T
evaluate: 2 is a number. push it to valStk.
# VAL STACK: B|14,4,3,2|T
#  OP STACK: B|<=,-,*|T
evaluate: + is an operator. Call repeatOps.
  repeatOps: Consume all Ops in opStack, which are higher or equal than +
  # VAL STACK: B|14,4,3,2|T
```

```
  #  OP STACK: B|<=,-,*|T
  repeatOps: + <= *
    # VAL STACK: B|14,4,3,2|T
    doOp: Pop 2, then 3 from valStk. Pop * from opStk.
    doOp: Calculate 3*2, and put it back to valStk.
    # VAL STACK: B|14,4,6|T
    #  OP STACK: B|<=,-|T
  repeatOps: + <= -
    # VAL STACK: B|14,4,6|T
    doOp: Pop 6, then 4 from valStk. Pop - from opStk.
    doOp: Calculate 4-6, and put it back to valStk.
    # VAL STACK: B|14,-2|T
    #  OP STACK: B|<=|T
  repeatOps: Because + > <=, it ends here
evalaute: push + into opStk
# VAL STACK: B|14,-2|T
#  OP STACK: B|<=,+|T
evaluate: 7 is a number. push it to valStk.
# VAL STACK: B|14,-2,7|T
#  OP STACK: B|<=,+|T
evalaute: End of the expression. Consume all the remaining operations
  repeatOps: Consume all Ops in opStack, which are higher or equal than $
  # VAL STACK: B|14,-2,7|T
  #  OP STACK: B|<=,+|T
  repeatOps: $ <= +
    # VAL STACK: B|14,-2,7|T
    doOp: Pop 7, then -2 from valStk. Pop + from opStk.
    doOp: Calculate -2+7, and put it back to valStk.
    # VAL STACK: B|14,5|T
    #  OP STACK: B|<=|T
  repeatOps: $ <= <=
    # VAL STACK: B|14,5|T
    doOp: Pop 5, then 14 from valStk. Pop <= from opStk.
    doOp: Calculate 14<=5, and put it back to valStk.
    # VAL STACK: B|False|T
    #  OP STACK: B||T
  repeatOps: Because there is only one value in valStk, it ends here.
```

Out[8]:

```
False
```

In [ ]:

```

```

In [ ]:

```

```