파이썬으로 배우는 데이터 구조

한동대학교
전산전자공학부
김영섭 교수

학습 목표

Postfix계산과 Infix to Postfix의

알고리즘을 이해하고 구현할 수 있다

Data Structures in Python
Chapter 3 - 1

- Stack Concept and ADT
- Stack Example - Matching
- **Stack Example - Postfix**

Agenda

- Postfix Calculation
- Conversion from Infix to Postfix

Infix, postfix and prefix expressions

Stacks can be used to implement algorithms involving Infix, postfix and prefix expressions.

- Infix:
  - An infix expression is a single letter, or an operator, proceeded by one infix string and followed by another infix string.
  - A, A + B, (A + B) + (C - D)
- Prefix:
  - A prefix expression is a single letter, or an operator, followed by two prefix strings. Every prefix string longer than a single variable contains an operator, first operand and second operand.
  - A, + A B, + + A B - C D
- Postfix:
  - A postfix expression (also called Reverse Polish Notation) is a single letter or an operator, preceded by two postfix strings. Every postfix string longer than a single variable contains first and second operands followed by an operator.
  - A, A B +, A B + C D - +

Infix, postfix and prefix expressions

- Prefix and postfix notations are methods of writing mathematical expressions without parenthesis.
  - **Why**: Time to evaluate a postfix and prefix expression is **O(n)**, where n is the number of elements in the array.

Infix	Prefix	Postfix
A + B	+ A B	A B +
A + B – C	– + A B C	A B + C –
(A + B) * C – D	– * + A B C D	A B + C * D –

Postfix Calculator

- Computation of arithmetic expressions can be efficiently carried out in Postfix notation with the help of stack.

infix		infix		postfix	Result
2 * 3 + 4	→	(2 * 3) + 4	→	2 3 * 4 +	10
		2 * (3 + 4)	→	2 3 4 + *	14

Postfix Calculator

- Requires you to enter postfix expressions.
  - Example: 2 3 4 + *

Algorithm:

- When an **operand** is entered,
  - the calculator pushes it onto a stack
- When an **operator** is entered,
  - the calculator applies it to the top **two operands** of the stack
    - Pops the top two operands from the stack
    - Pushes the result of the operation on the stack

Postfix Calculator - Algorithm

- Example 1: Evaluating the expression: 2 3 4 + *

Key entered	Calculator action		Stack(bottom to top)
2	push 2		2
3	push 3		2 3
4	push 4		2 3 4
+	operand2 = pop stack	(4)	2 3
	operand1 = pop stack	(3)	2
	result = operand1 + operand2	(7)	2
	push result		2 7
*	operand2 = pop stack	(7)	2
	operand1 = pop stack	(2)	
	result = operand1 * operand2	(14)	
	push result		14

Postfix Calculator

- Example 2: Evaluating the expression: 2 3 * 4 +

Key entered	Calculator action		Stack(bottom to top)
2	push 2		2
3	push 3		2 3
*	operand2 = pop stack	(3)	2
	operand1 = pop stack	(2)	
	result = operand1 * operand2	(6)	
	push result		6
4	push 4		6 4
+	operand2 = pop stack	(4)	6
	operand1 = pop stack	(4)	
	result = operand1 + operand2	(10)	
	push result		10

Postfix Calculator

- Example 3: Evaluating the expression: 12 3 - 3 /

Key entered	Calculator action		Stack(bottom to top)
12	push 12		12
3	push 3		12 3
-	operand2 = pop stack	(3)	12
	operand1 = pop stack	(12)	
	result = operand1 + operand2	(9)	
	push result		9
3	push 3		9 3
/	operand2 = pop stack	(3)	6
	operand1 = pop stack	(9)	
	result = operand1 / operand2	(3)	
	push result		3

The order of operand1 and operand2 is very important.

Postfix Calculator - Exercise 2

- Evaluate the expression: 10 4 2 - 5 * + 3 -

Key entered	Calculator action	Stack(bottom to top)

Postfix Calculator

- Coding

```
def evaluate_postfixList(postfixList):
    stack = Stack()
    operators = "+-/*"
    for op in postfixList:
        if op in operators:    #operator
            if stack.size() > 1:
                num2 = stack.pop()
                num1 = stack.pop()
                result = compute(int(num1), int(num2), op)
                stack.push(result)
            else:
                return "Failed while parsing postfix expression"
        else: #operand
            stack.push(op)
    return stack.pop()
```

Write your own compute() function to make this code work properly.

```
#Sample Run:
evaluate_postfixList(['3', '4', '7', '*', '+'])
31
```

Conversion from Infix to Postfix

- Examples:
  - 2 * 3 + 4 → 2 3 * 4 +
  - 2 + 3 * 4 → 2 3 4 * +
  - 1 * 3 + 2 * 4 → 1 3 * 2 4 * +

- Algorithm Concept:
  - Operands always stay in the same order with respect to one another.
  - An operator will move only “to the right” with respect to the operands.
  - All parentheses are removed.

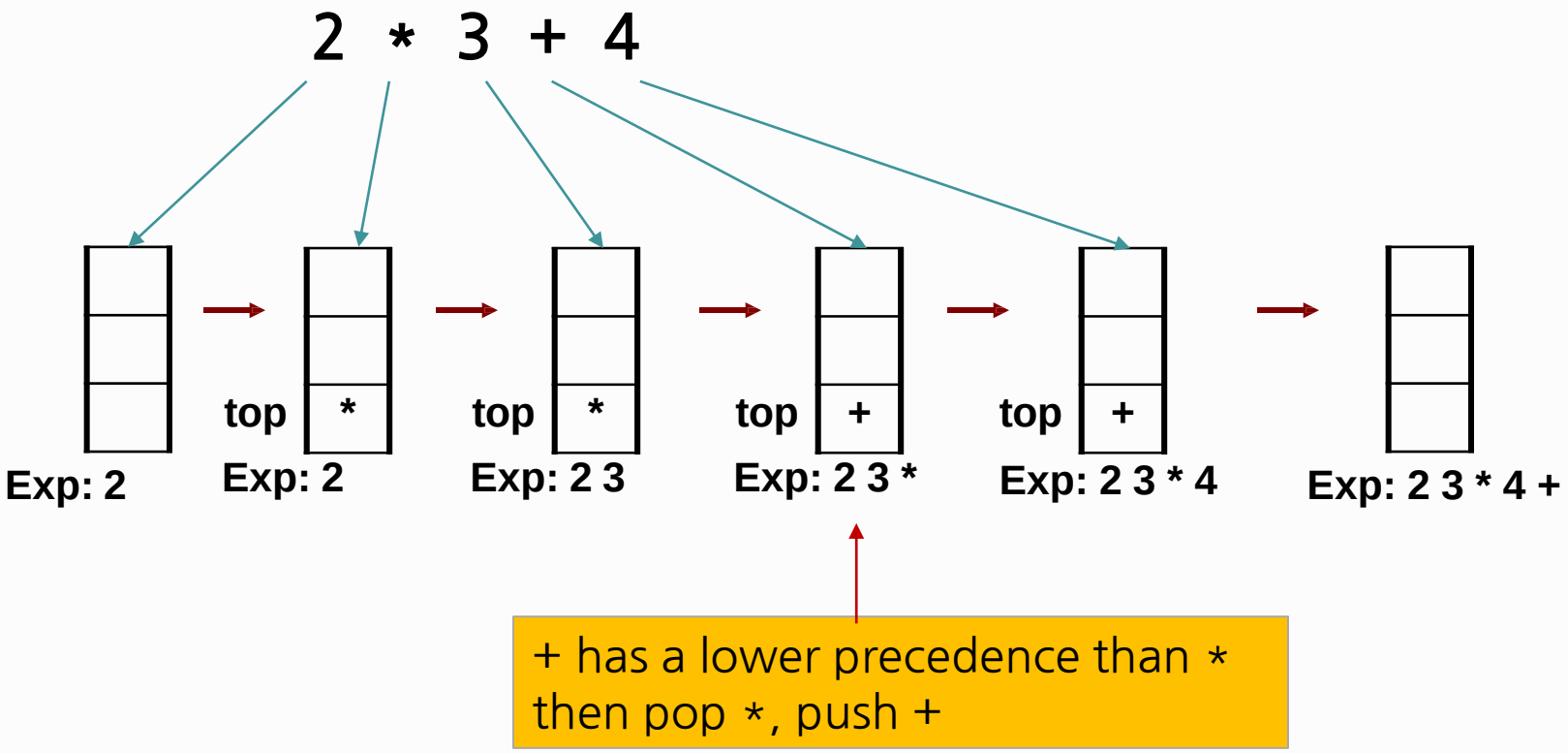
Conversion from Infix to Postfix

Algorithm:

- operand - output it to postfixExp.
- “(“ - push onto the stack.
- “)” - pop the operators off the stack and append them to the end of postfixExp until encounter the match “(“.
- operator
  - For **higher** precedence operator, push it onto the stack.
  - For **lower or equal** precedence operator, pop them until "(" and push it onto the stack.
- End of the string
  - append the remaining contents of the stack to postfixExp.

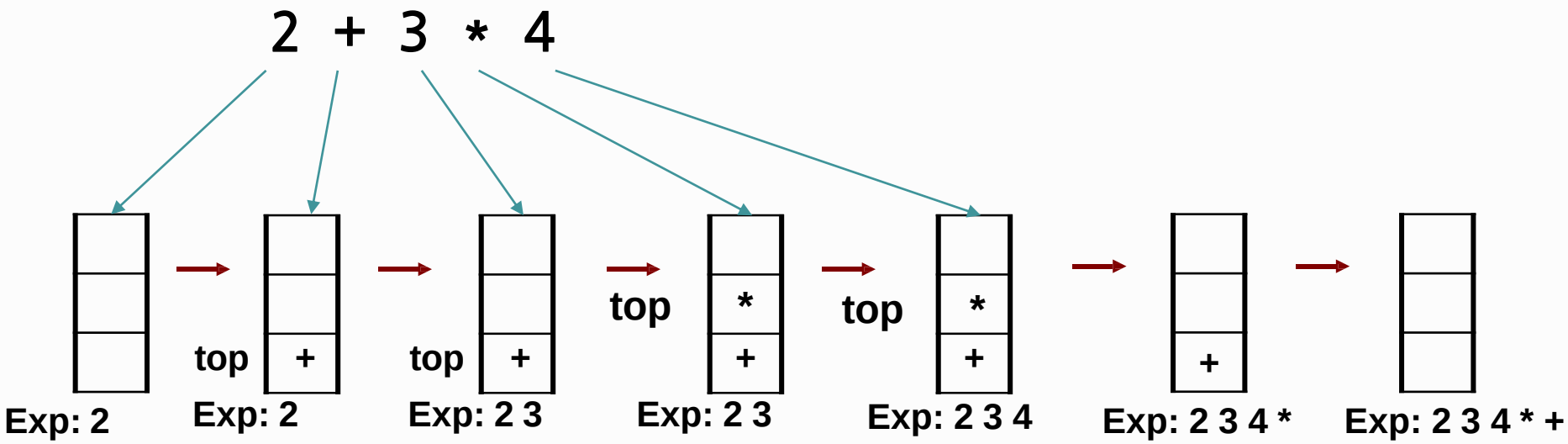
Conversion from Infix to Postfix

- Example 1



Conversion from Infix to Postfix

- Example 2:



top +

Conversion from Infix to Postfix

- Example 3: a - (b + c * d) / e

token	stack	postfix	
a		a	
-	-	a	
(	-(	a	
b	-(	ab	
+	-(+	ab	
c	-(+	abc	
*	-(+*	abcd	
)	-(+	abcd*	get operators
	-(	abcd*+	from stack to
	-	abcd*+	postfix until "("
/	-/	abcd*+	
e	-/	abcd*+e	get operators
			from stack to
			postfix until empty

bottom top

Conversion from Infix to Postfix - Exercise 3

- Debug the following program.

```
def get_postfix(infixList):
    precedence = {"*":3, "/":3, "+":2, "-":2, "(":1 }
    operators = "+-/*"
    op_stack = Stack()
    postfixList = []
    for op in infixList:
        if op in operators:
            while (not op_stack.is_empty()) and (precedence[op_stack.peek()] >= precedence[op]):
                postfixList.append(op_stack.pop())
                op_stack.push(op)
        elif op == "(":
            op_stack.push(op)
        elif op == ")":
            op = op_stack.pop()
            while not op == "(":
                postfixList.append(op)
            op = op_stack.pop()
        else: #operand
            postfixList.append(op)
    while not op_stack.is_empty():
        postfixList.append(op_stack.pop())
    return " ".join(postfixList), postfix
```

```
#Sample Run:
a, b = get_postfixList(['3', '+', '4', '*', '7'])
print(a)
print(b)
```

```
3 4 7 * +
['3', '4', '7', '*', '+']
```

Conversion from Infix to Postfix - Exercise 4

- Converting the infix expression to postfix: (B - C) * (D - E)

token	stack	postfix

Conversion from Infix to Postfix - Exercise 4 solution

- Converting the infix expression to postfix: (B - C) * (D - E)

token	stack	postfix
(	(	
B		B
-	(-	B
C	(-	BC
)	(	BC-
	)	BC-
*	*	BC-
(	*(	BC-
D	*(	BC-D
-	*(-	BC-D
E	*(-	BC-DE
)	*(	BC-DE-
	*	BC-DE-
		BC-DE-*

```
a, b = get_postfix(['(', 'B', '-', 'C', ')', '*', '(', 'D', '-', 'E', ')'])
print(a)
print(b)
```

```
3 4 7 * +
['3', '4', '7', '*', '+']
```

Summary

- Stacks are used in applications that manage data items in LIFO manner, such as:
  - Checking for Balanced Braces
  - Matching bracket symbols in expressions
  - Evaluating postfix expressions
  - Conversion from Infix to Postfix

학습 정리

1) 전위(Prefix) 및 후위(Postfix) 표현식에는 괄호 연산이
없기 때문에 Infix(중위) 표현식보다 효율적으로 계산할 수 있다

2) Infix to Postfix 표현식 변환 알고리즘의 핵심은 괄호를
모두 제거하고 연산자(operator)에 우선순위를 부여하는 것이다

파이썬으로 배우는 데이터 구조

수고했습니다
곧 다음 시간에
다시 뵙겠습니다