COMPSCI 105 S1 2017
Principles of Computer Science

14 Stack (2)

Agenda & Readings

- Agenda
  - Example Applications
    - Checking for Balanced Braces
    - Bracket Matching
    - Postfix Calculation
    - Conversion from Infix to Postfix
  - Reference:
    - Textbook: Problem Solving with Algorithms and Data Structures
      - Chapter 3: Basic Data Structures

Checking for Balanced Braces

- Using a stack to verify whether braces are balanced
  - An example of balanced braces
    - {{}{{}}}
  - An example of unbalanced braces
    - {}}{{}
  - Requirements for balanced braces
    - Each time you encounter a "}", it matches an already encountered "{"
    - When you reach the end of the string, you have matched each "{"

Balanced Braces - Algorithm

- Steps

Initialise the stack to empty

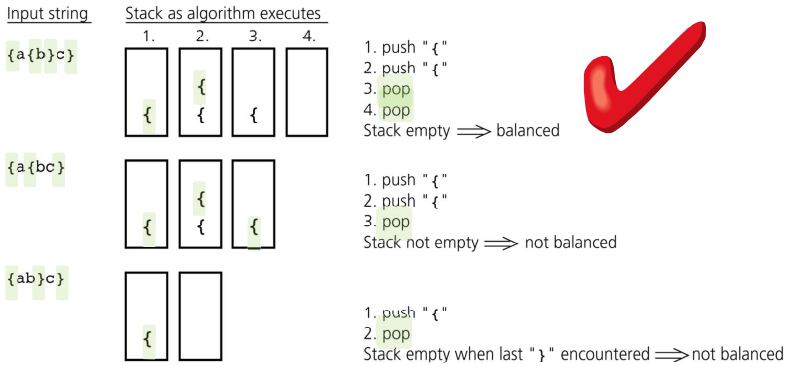
For every char read

- If it is an open bracket then push onto stack
- If it is a close bracket,
  - If the stack is empty, return ERROR
  - Else, pop an element out from the stack
- if it is a non-bracket character, skip it

If the stack is NON-EMPTY, ERROR

Balanced Braces - Algorithm

- Examples



Coding

- Codes and results

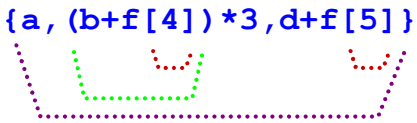
```
def braces_are_balanced(expression_to_test):
    st = Stack()
    for i in range(len(expression_to_test)):
        letter = expression_to_test[i]
        if letter == '{':      #open bracket
            st.push(letter)
        elif letter == '}':   #close bracket
            if st.is_empty():
                return False
            else:
                st.pop()        #pop an item
    return st.is_empty()
```

```
expression_to_test
{a{b}c}
{a{bc}
{ab}c}

Result:
{a{b}c}: True
{a{bc}: False
{ab}c}: False
```


Bracket Matching

- Ensure that pairs of brackets are properly matched
  - Examples:



Bracket Matching

- Ensure that pairs of brackets are properly matched
  - Other Examples:

```
(..)..)      // too many closing brackets

(..(..)      // too many open brackets

[..(..]..)   // mismatched brackets
```

Bracket Matching - Algorithm

- Steps

Initialise the stack to empty

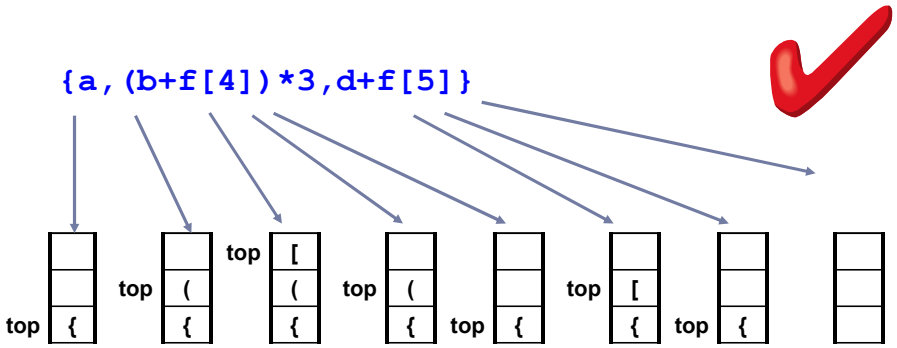
For every char read

- if it is an open bracket then push onto stack
- if it is a close bracket, then
  - If the stack is empty, return ERROR
  - pop from the stack
  - if they don't match then return ERROR
- if it is a non-bracket, skip the character

If the stack is NON-EMPTY, ERROR

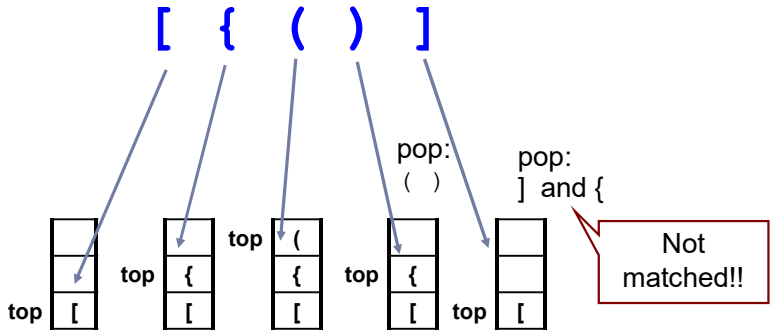
Bracket Matching - Algorithm

- Example 1



Bracket Matching - Algorithm

- Example 2



Coding

- Codes and results

```
def brackets_checker(symbol_string):
    st = Stack()
    balanced = True
    index = 0
    while index < len(symbol_string) and balanced:
        symbol = symbol_string[index]
        if symbol in "([{":
            st.push(symbol)
        elif symbol in ")]}":
            if st.is_empty():
                balanced = False
            else:
                top = st.pop()
                if not matches(top, symbol):
                    balanced = False
        index = index + 1
    if balanced and st.is_empty():
        return True
    else:
        return False
```

A function to check whether the brackets are matched

Example:
[{()]
[{()}]
[{()}]}

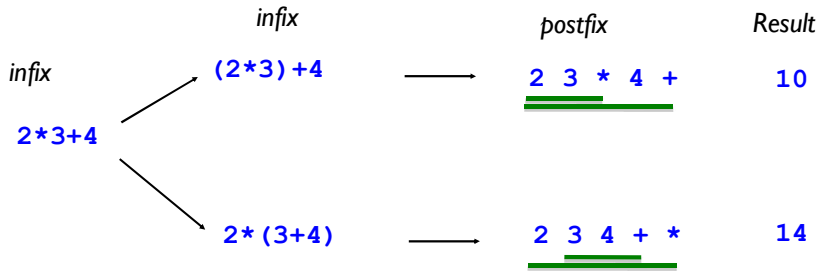
Result:
[{()]: False
[{()}] :True
[{()}]} : False

Exercise 1

- Complete the function match(a, b)
  - It is a function to check whether the brackets are match
    - Examples: a = '('; b = ')' return True

Postfix Calculator

- Computation of arithmetic expressions can be efficiently carried out in Postfix notation with the help of stack
  - Infix - arg1 op arg2
  - Prefix - op arg1 arg2
  - Postfix - arg1 arg2 op



Postfix Calculator - Algorithm

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

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: 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 - Algorithm

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

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

Exercise 2

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

Postfix Calculator - Algorithm

- Example 2: Evaluating the expression: 12 3 - 3 /

Key entered	Calculator action	Stack
12	push 12	
3	push 3	
-	operand2 = pop Stack (3) operand1 = pop Stack (12) result = operand1 - operand2 (9) push result	
3	push 3	
/	operand2 = pop Stack (3) operand1 = pop Stack (9) result = operand1 / operand2 (3) push result	

Order is very important

Coding

- Codes and results

A function to compute the corresponding operation

```
def evaluate_postfix_expression(postfix_i
    expression_stack = Stack()
    allowed_operators = "+-/*"

    for i in postfix_in_list:
        if i in allowed_operators: #operator
            if expression_stack.siz  > 1:
                number2 = expre ion_stack.pop()
                number1 = exp ession_stack.pop()
                result = compute(int(number1), int(number2), i)
                expression_stack.push(result)
            else:
                return "Error while parsing postfix expression"
        else: #operand
            expression_stack.push(i)

    return expression_stack.pop()
```

Examples:
2 3 * 4 +
2 3 4 * +
12 3 - 2 /

Result: :
10
14
4.5

Conversion from Infix to Postfix

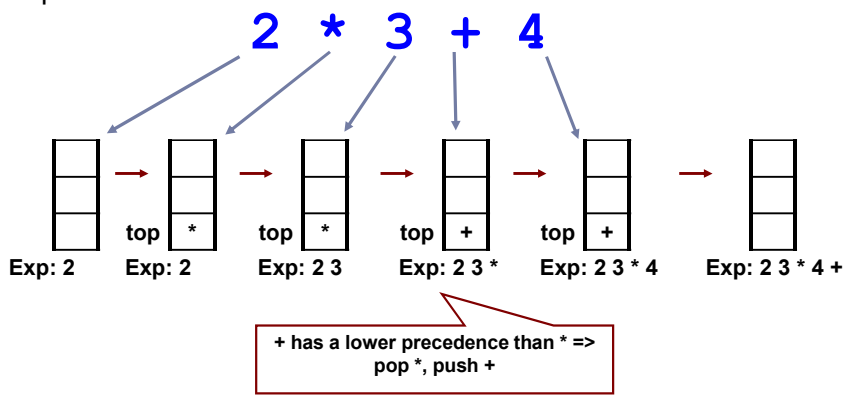
- Examples:
  - 2 * 3 + 4 => 2 3 * 4 +
  - 2 + 3 * 4 => 2 3 4 * +
  - 1 * 3 + 2 * 4 => 1 3 * 2 4 * +
- Steps:
  - Operands always stay in the same order with respect to one another
  - An operator will move only "to the right" with respect to the operands
  - All parentheses are removed

Conversion - Algorithm

- operand – append it to the output string postfixExp
- "(" – push onto the stack
- operator
  - If the stack is empty, push the operator onto the stack
  - If the stack is non-empty
    - Pop the operators of greater or equal precedence from the stack and append them to postfixExp
    - Stop when encounter either a "(" or an operator of lower precedence or when the stack is empty
    - Push the new operator onto the stack
- ")" – pop the operators off the stack and append them to the end of postfixExp until encounter the match "("
- End of the string – append the remaining contents of the stack to postfixExp

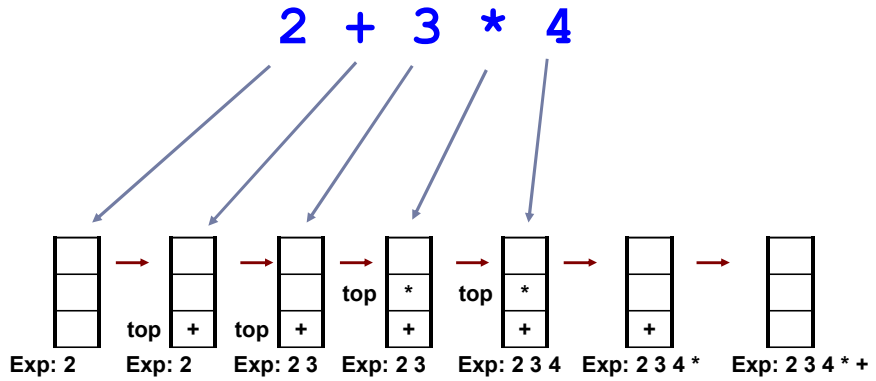
Conversion - Algorithm

- Example 1



Conversion - Algorithm

- Example 2



Conversion - Algorithm

- Example 3: a – (b + c * d) / e

ch	stack (bottom to top)	postfixExp	
a		a	
–	–	a	
(	– (	a	
b	– (	ab	
+	– (+	ab	
c	– (+	abc	
*	– (+ *	abc	
d	– (+ *	abcd	
)	– (+	abcd*	Move operators
	– (	abcd*+	from stack to
	–	abcd*+	postfixExp until " ("
/	– /	abcd*+	
e	– /	abcd*+e	Copy operators from
		abcd*+e/–	stack to postfixExp

Coding

- Codes: e.g. 3 + 4 * 7 => 3 4 7 * +

```
def get_postfix_expression(infix_list):
    prec_dictionary = {"*":3, "/":3, "+":2, "-":2, "(":1 }
    allowed_operators = "+-/*"
    op_stack = Stack()
    postfix_list = []
    for i in infix_list:
        if i in allowed_operators:
            while (not op_stack.is_empty()) and (prec_dictionary[op_stack.peek()] >= prec_dictionary[i]):
                postfix_list.append(op_stack.pop())
            op_stack.push(i)
        elif i == "(":
            op_stack.push(i)
        elif i == ")":
            i = op_stack.pop()
            while not i == "(":
                postfix_list.append(i)
                i = op_stack.pop()
        else: #operand
            postfix_list.append(i)
    while not op_stack.is_empty():
        postfix_list.append(op_stack.pop())
    s = " "
    return s.join(postfix_list)
```


Exercise 3

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

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