Stacks

Thomas Schwarz, SJ

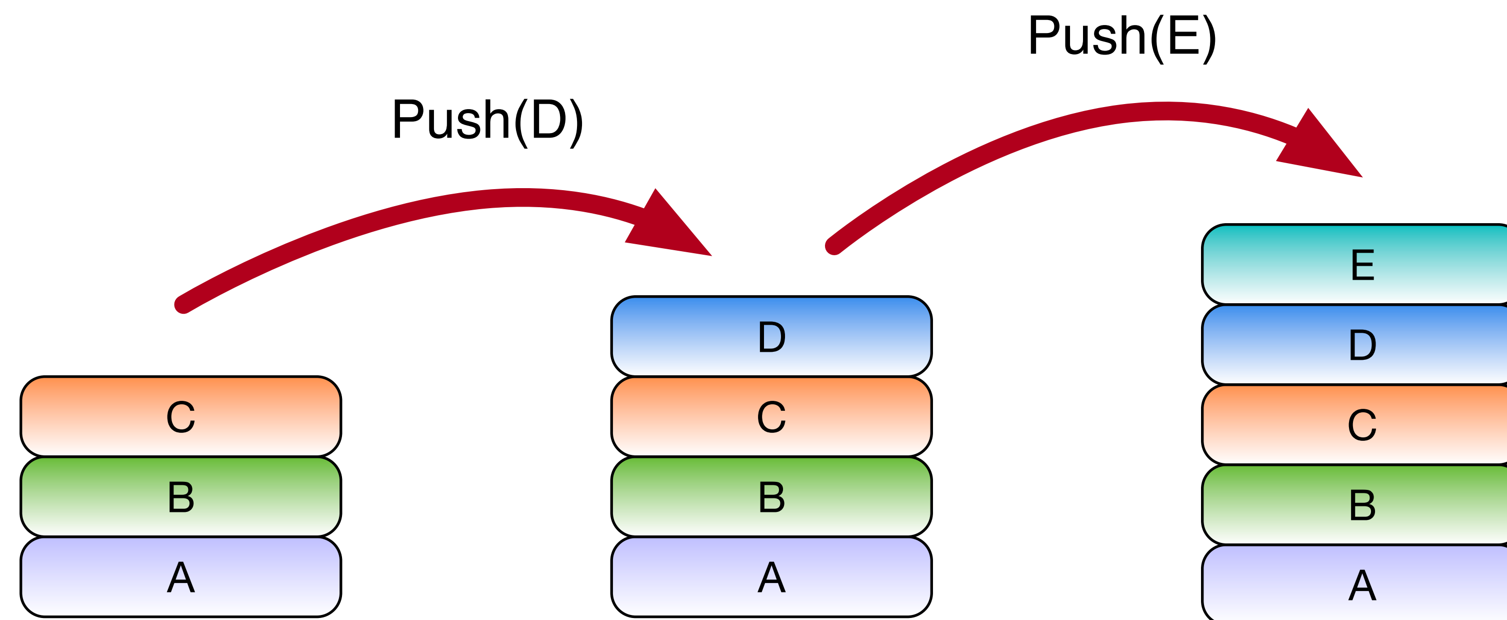
Stacks

- Stacks:
  - Named after a stack of papers or a stack of dishes in a cafeteria

  - When you put a dish on the stack, it slightly goes down
  - When you take a dish off the stack, the stack slightly goes up

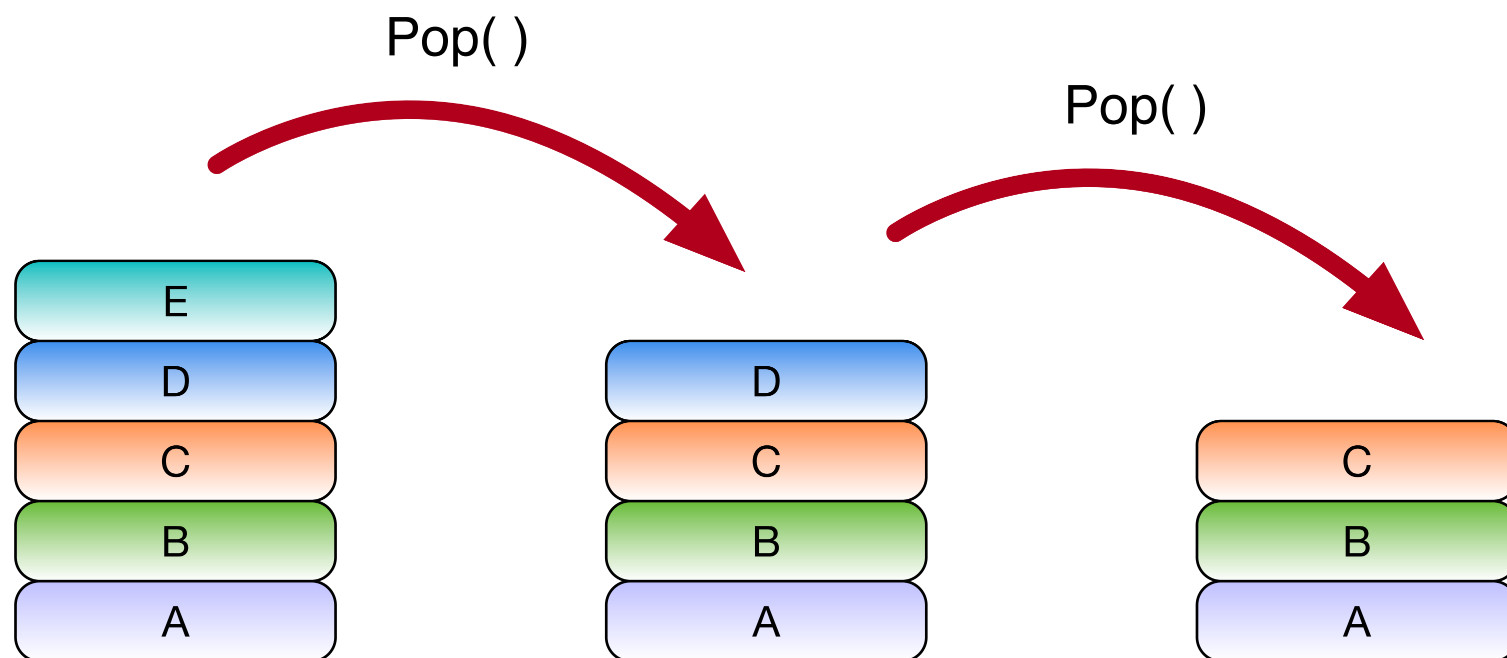
Stacks

- Stack a.k.a. LIFO queue
    - Last In -- First Out
- Two operations:
    - Push(object)



Stacks

- Two operations:
  - Pop( ) returns uppermost object



Stacks

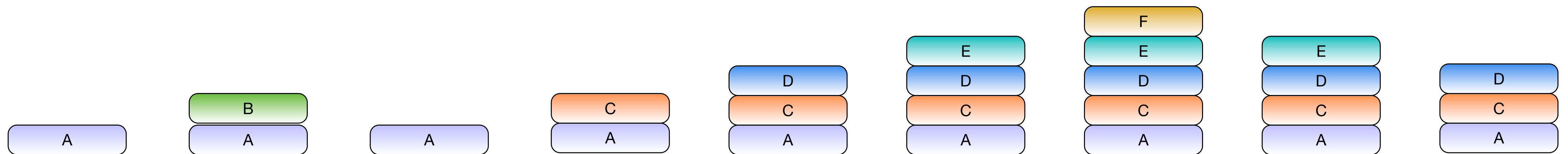
- Your turn:
  - An initially empty stack
    - What is the content of the stack after the following series of operations

```
push(A), push(B), pop( ), push(C)
push(E), push(F), pop( ), pop( )
```

    - And what is the result of the last operation?

Stacks

push(A), push(B), pop( ), push(C)
push(E), push(F), pop( ), pop( )



Last pop returns E

Stacks

- There are two auxiliary methods:
  - peek( )
    - Look at the last element pushed on a stack
    - Raises error or returns empty is stack is empty
    - Does not change the stack
  - is_empty( )
    - returns True if the stack is empty

Python Implementation

- Can use a Python list
  - pop( ) removes the last element
  - append( ) adds at the end

Python Implementation

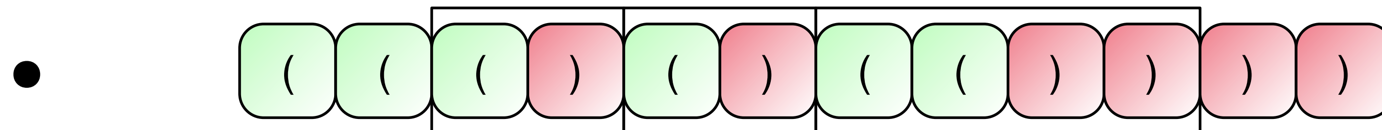
- Use an internal component that is a Python list

```
class Stack:
    def __init__(self):
        self.array = []
    def peek(self):
        return self.array[-1]
    def pop(self):
        return self.array.pop()
    def push(self, value):
        self.array.append(value)
    def is_empty(self):
        return len(self.array)==0
    def __str__(self):
        return str(self.array)
```

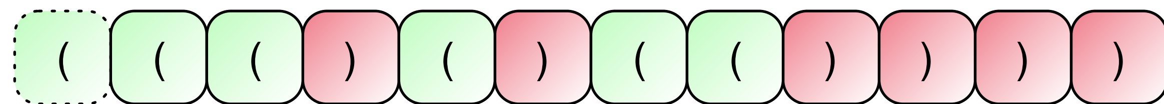
Parenthesization

- Is an expression with parentheses well balanced?
  - E.g. well-balanced: (( ) ( )) ((( )))
  - Not well-balanced: ((( ))

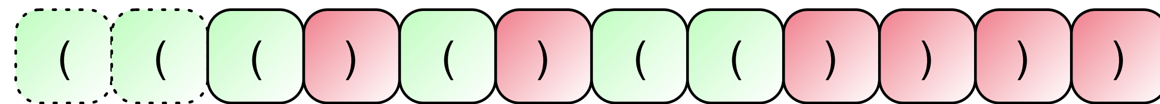
Parenthesization

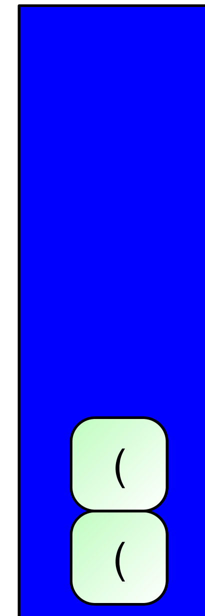
- Example:
  - 

- Process from left to right
- Push on stack



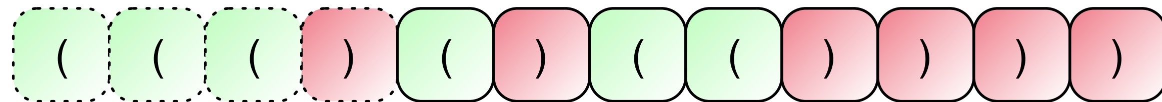
Parenthesization

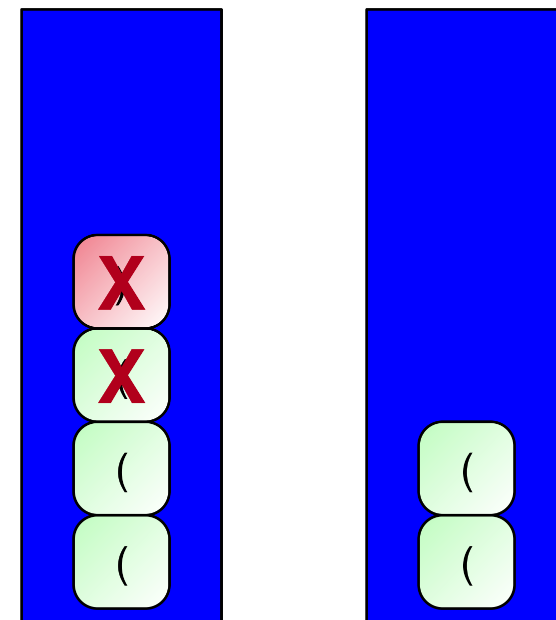




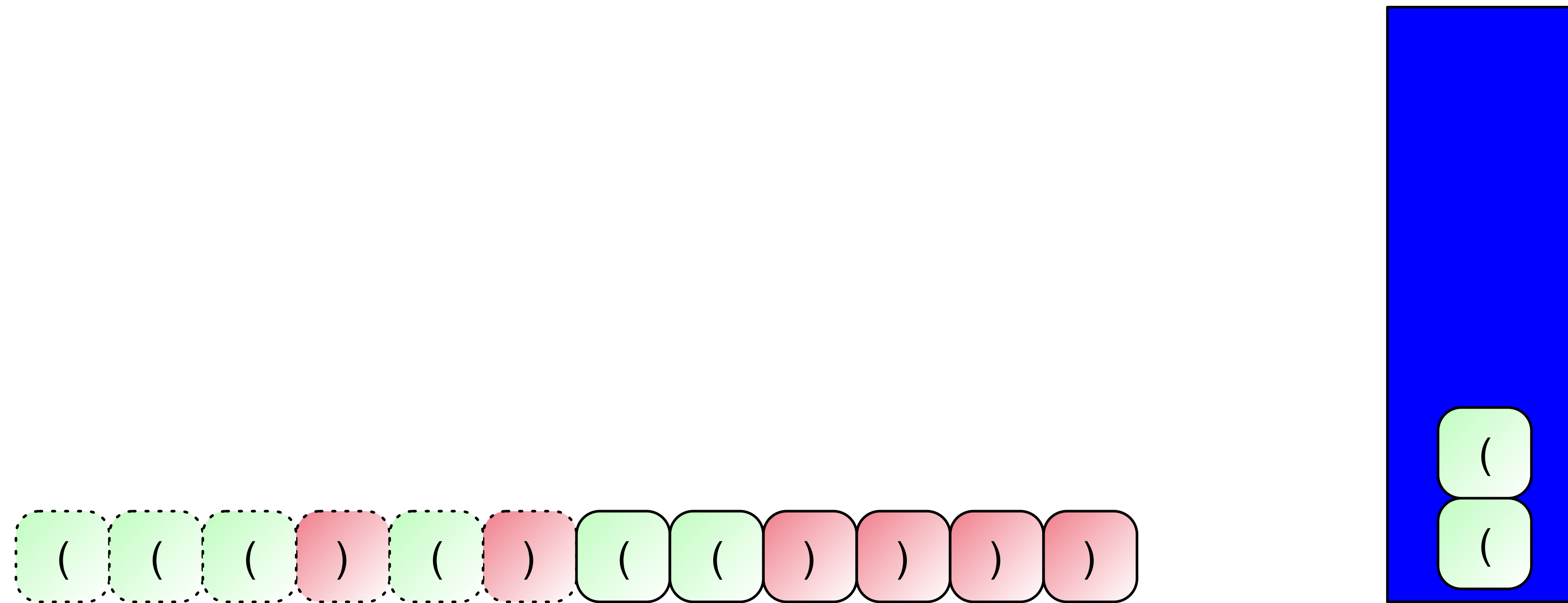
Parenthesization

- When we push a closing parenthesis, we see whether we can pop by combining with an opening parenthesis

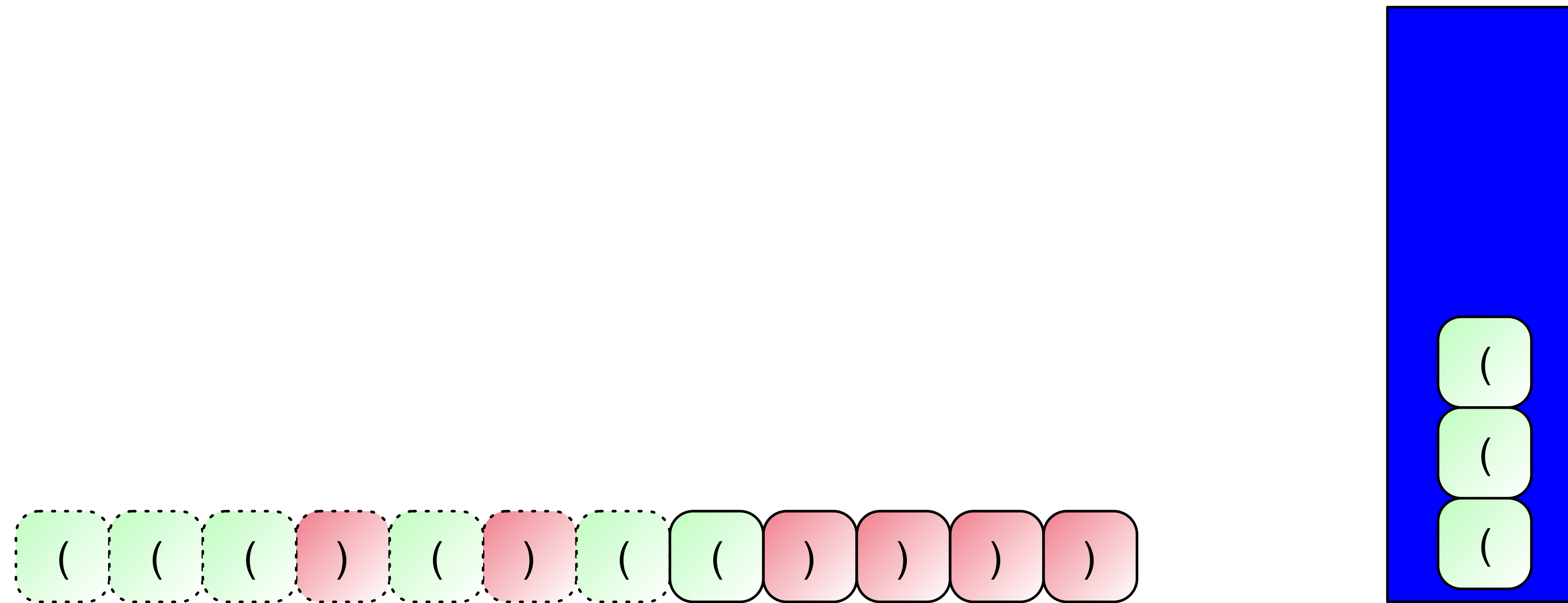




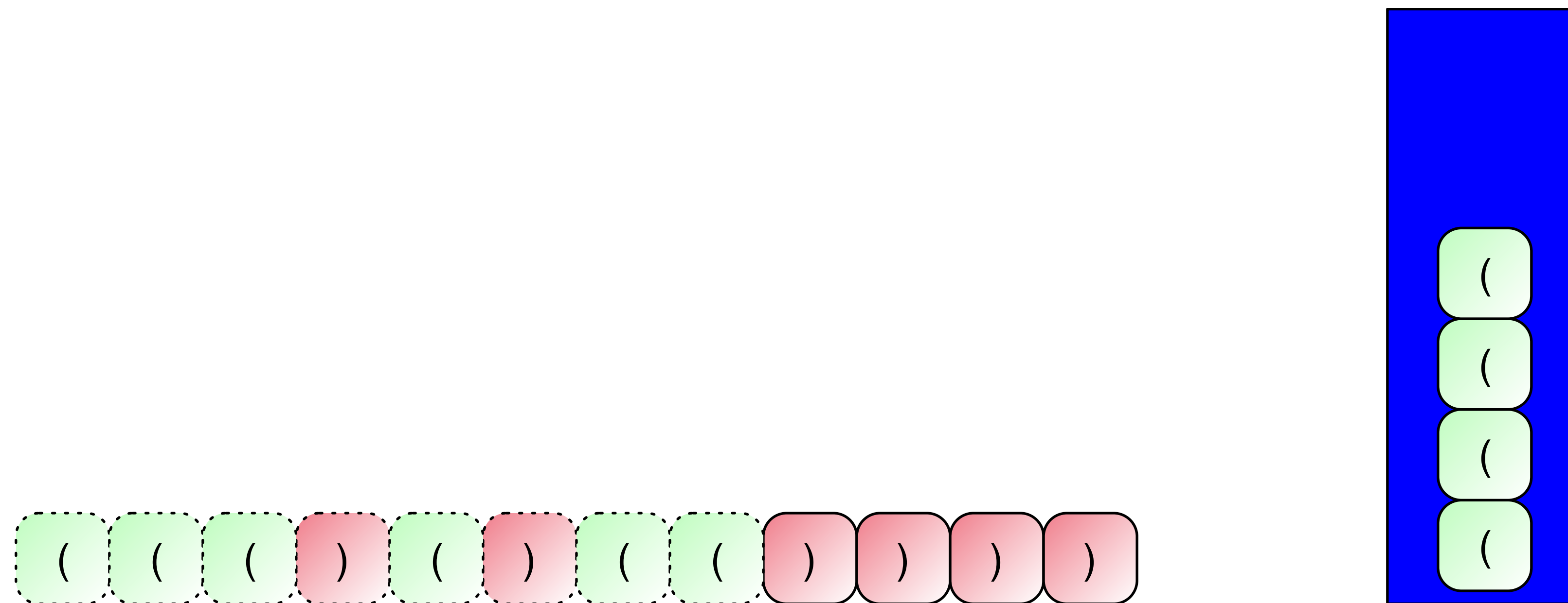
Parenthesization



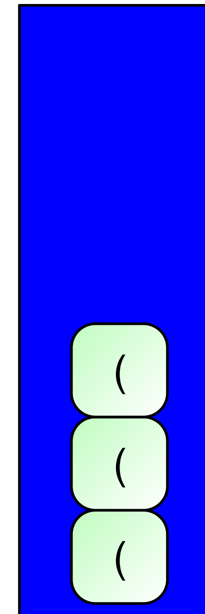
Parenthesization

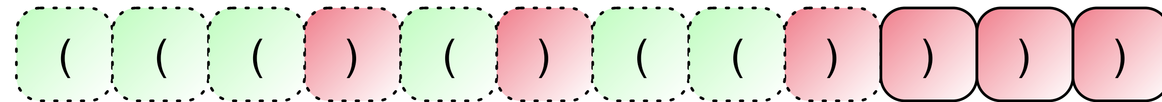


Parenthesization

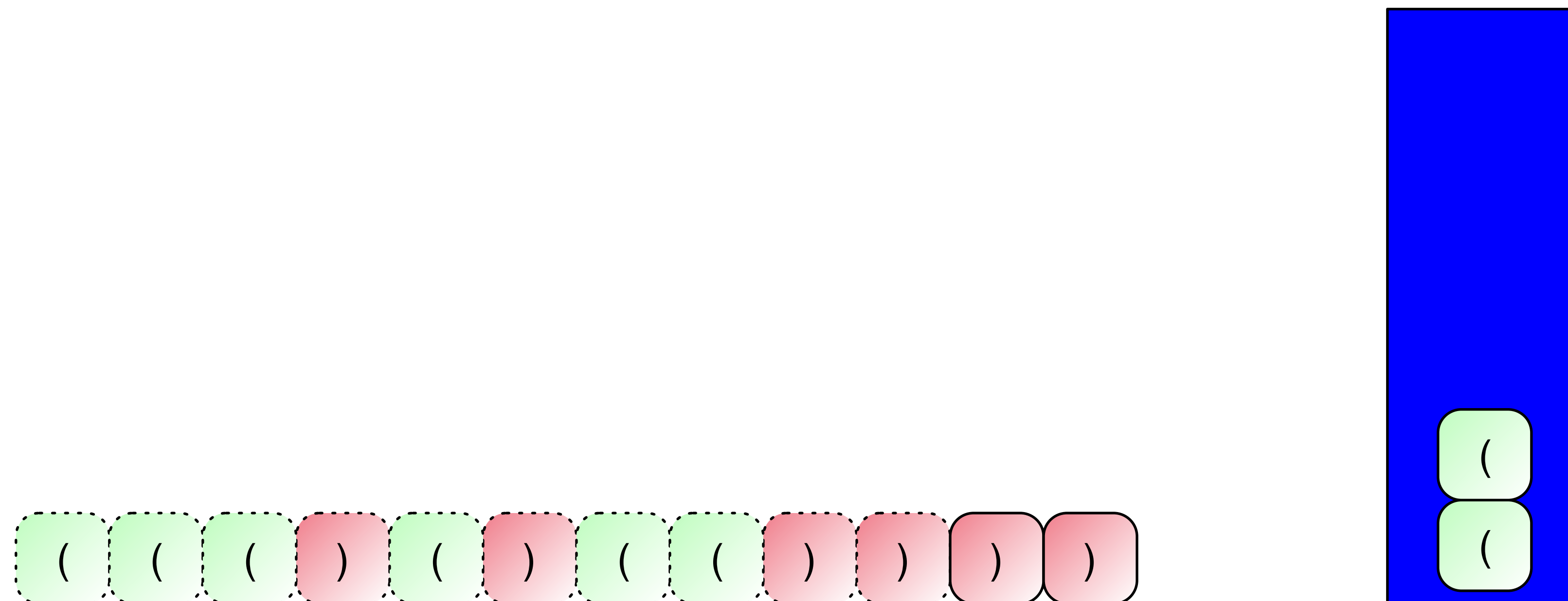


Parenthesization



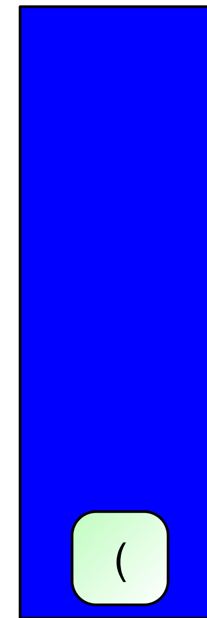


Parenthesization

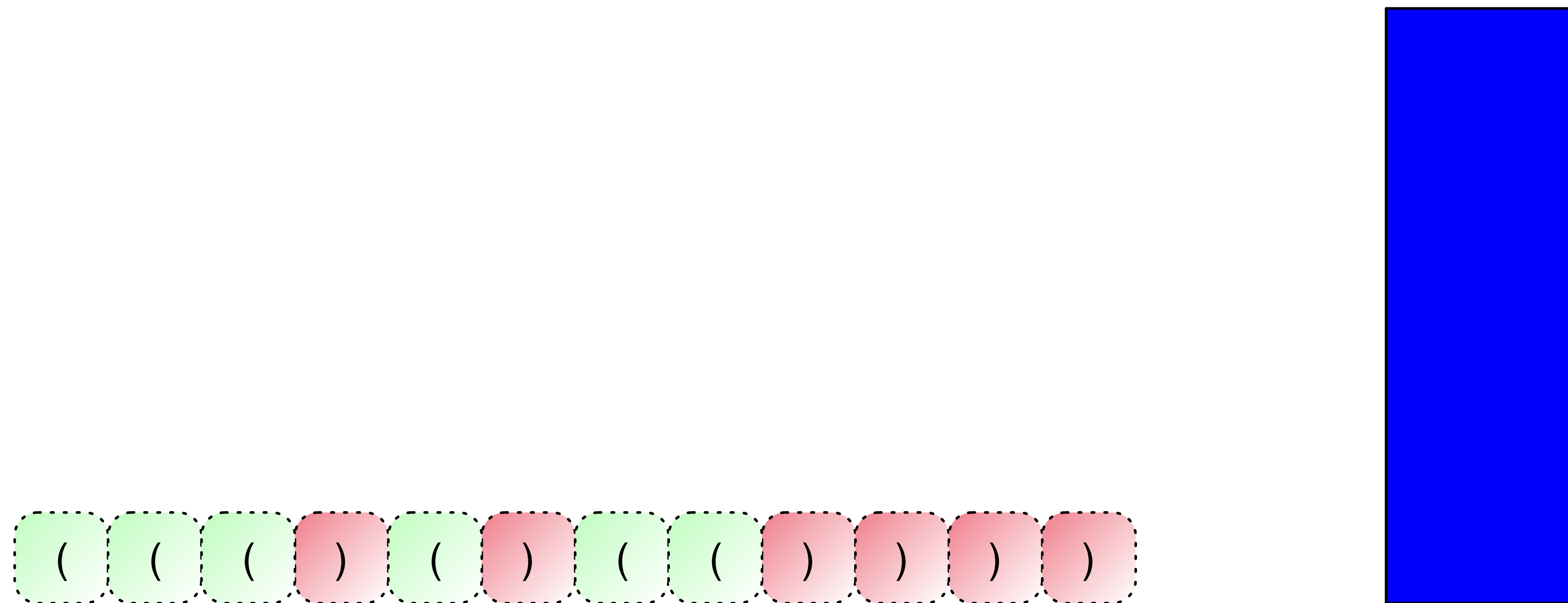


Parenthesization

((( ) ( ) (( ))))



Parenthesization



The stack is empty, the string has been accepted

Parenthesization

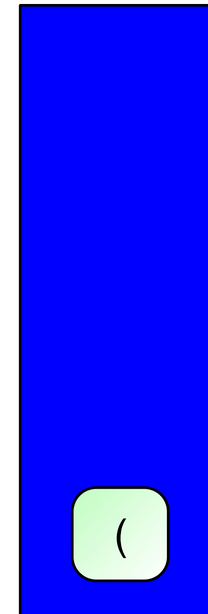
- This can be extended to several types of parentheses
  - Example: "[( ]) ( )" is malformed

Parenthesization

([ )] ( )

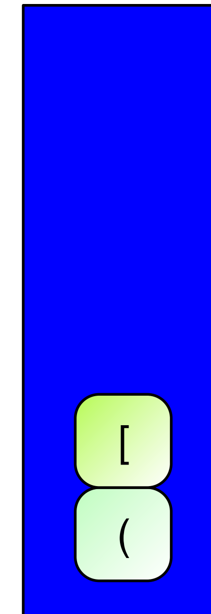
Parenthesization

([ )] ( )

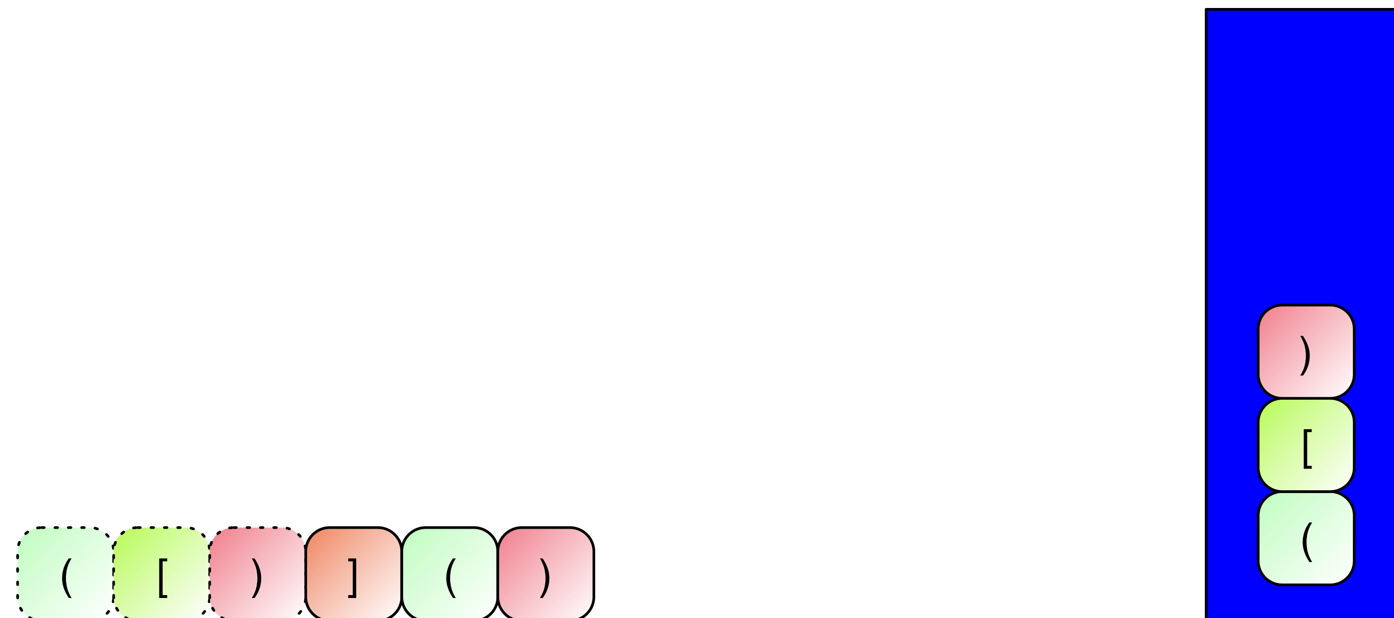


Parenthesization

([ )] ( )



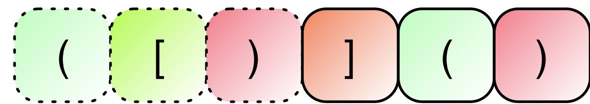
Parenthesization

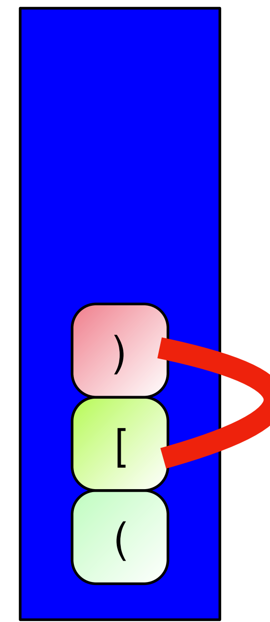


We try to push a ')' but cannot match it with previous one
This is how we can recognize a bad parenthesization

Parenthesization

- When-ever a closing parenthesis is encountered:
  - We try to match
    - In which case we pop the other part of the pair
  - And if that is not possible
    - We declare defeat





conflict

Parenthesization

- Implementation
  - Interface:
    - Use the keyboard to enter parentheses, one at a time
    - Use 'Stop' to indicate the end of the expression

```
while True:
      inp = input(': ')
      if inp == 'Stop':
                  return
```

Parenthesization

- Creating an internal stack importing stack
- Processing 'Stop'
    - Expression is well-formed if the stack is empty

```
import stack

def test():
    my_stack = stack.Stack()
    while True:
        print(my_stack)
        inp = input(': ')
        if inp == 'Stop':
            return my_stack.is_empty()
```

Parenthesization

- Process parentheses:
    - Opening brackets are pushed

```
def test():
    my_stack = stack.Stack()
    while True:
        print(my_stack)
        inp = input(': ')
        if inp == 'Stop':
            return my_stack.is_empty()
        elif inp == '(':
            my_stack.push('(')
        elif inp == '[':
            my_stack.push('[')
```

Parenthesization

- Closing brackets are processed
  - If the stack is empty:
    - Expression not balanced
  - If top element matches:
    - Pop the top element
  - If the top element does not match
    - Expression not balanced

```
elif inp == ')':
    if my_stack.is_empty():
        return False
    elif my_stack.peek( ) == '(':
        my_stack.pop( )
    else:
        return False
```

Reverse Polish Notation

- Polish Notation was invented to show that arithmetical expressions can be expressed without parentheses
- Reverse Polish Notation (RPN) used in calculators until rather recently
    - Central idea: Enter operands, then the operation
-

Reverse Polish Notation

- Example:
  - $(3 + 4) * (5 - 7) + 1$
  - Take operands, place operator afterwards
    - $(3 + 4) * (5 - 7),1,+$
    - $(3 + 4),(5 - 7), * ,1,+$
    - $3,4, + ,5,7 - , * ,1,+$
  - Notice that we are using commata to separate constituents of the expression

Reverse Polish Notation

- Example:
  - $1, 2, 3, 4, +, 5, -, 6, *, /, +$
  - $1, 2, 3 + 4, 5, -, 6, *, /, +$
  - $1, 2, 3 + 4, 5, -, 6, *, /, +$
  - $1, 2, ((3 + 4) - 5), 6, *, /, +$
  - $1, 2, (((3 + 4) - 5) \cdot 6), /, +$
  - $1, \frac{2}{((3 + 4) - 5) \cdot 6}, +$
  - $1 + \frac{2}{((3 + 4) - 5) \cdot 6}$

Reverse Polish Notation

- We can use a stack to evaluate an arithmetic expression in RPN
  - Processing $3,5,+,7,2,-,*$ from left to right
  - Push 3 then 5 on a stack [3,5]
  - When processing the '+' operator, pop the last two from the stack, add them and push the result [8]
  - Push 7, then 2 on the stack [8,7,2]
  - When processing the '-' operator, pop the last two and push the difference [8,5]
  - Processing the '*' operator: Pop the last two and push the product [40]
  - This is the result

Reverse Polish Notation

- Assume we are given an expression in RPN
  - Separated by spaces
  - Return None plus emit error message if expression is malformed

Reverse Polish Notation

- We first take the string and split it (around white spaces)
- We then separately handle operators and integers
- If we try to pop from an empty stack, we know that the expression is malformed

```
def rpn(a_string):
    my_stack = stack.Stack()
    components = a_string.split()
    print(components)
    for x in components:
        if x in ['+', '-', '*', '/']:
            ...
        else:
            ...
```

Reverse Polish Notation

- An operator:
  - Try to pop twice and then push the result of the operation applied on the popped numbers

```
if x in ['+', '-', '*', '/']:
            try:
                a = my_stack.pop()
                b = my_stack.pop()
                if x == '+':
                    my_stack.push(a+b)
                elif x == '-':
                    my_stack.push(b-a)
                elif x == '*':
                    my_stack.push(a*b)
                elif x == '/':
                    my_stack.push(b/a)
            except IndexError:
                print('Expression malformed', x, my_stack)
                return None
```

Reverse Polish Notation

- If the component is not an operator:
  - Make it into a number
  - And push it

```
else:
            try:
                a = int(x)
            except ValueError:
                print('Expression malformed', x)
                return None
            my_stack.push(a)
```

Reverse Polish Notation

- At the end: the stack should house a single number
  - Pop it
- Nota bene: The stack cannot be empty at this point
- But it could have more than one number
  - Deal with this as homework