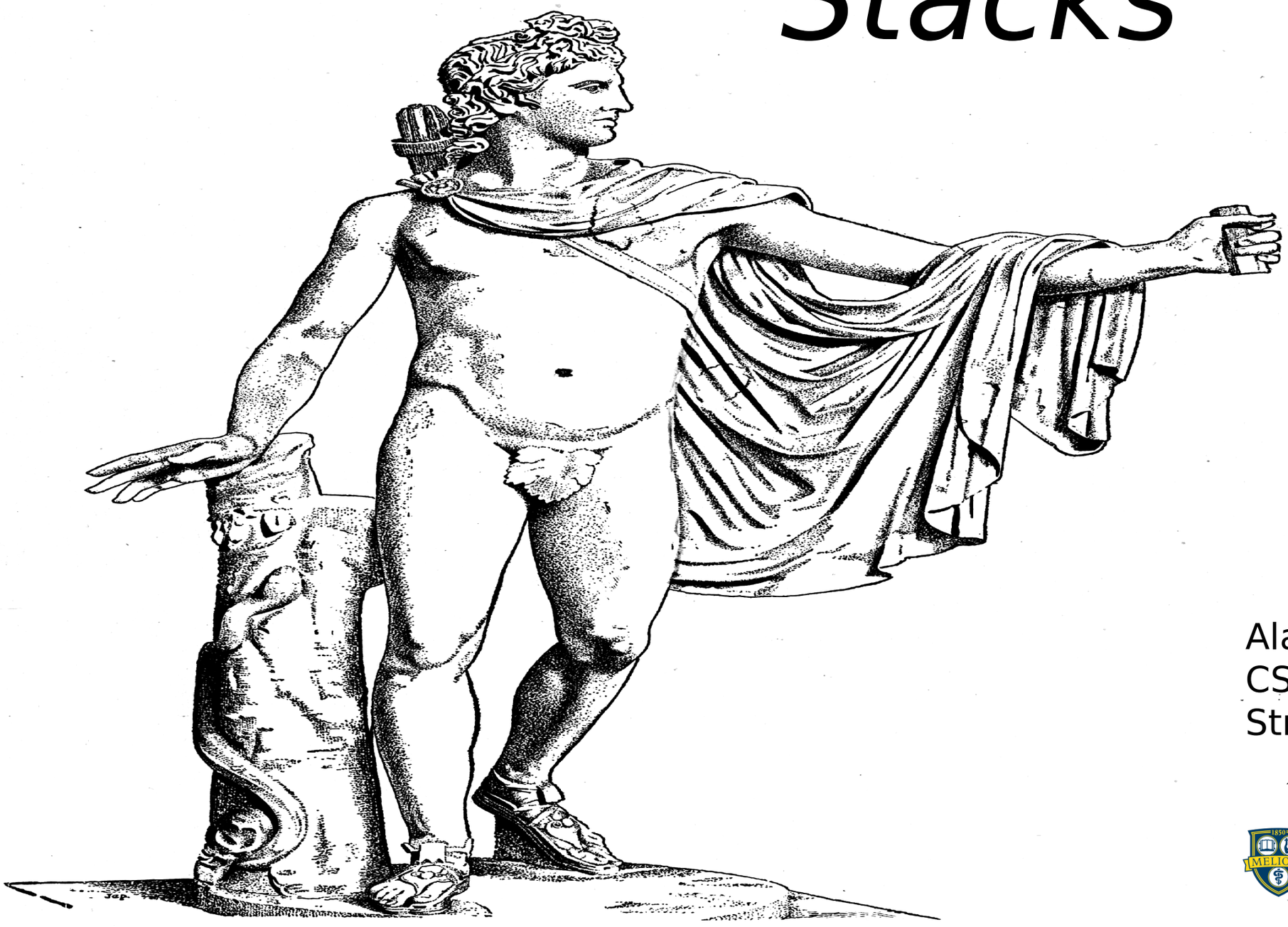
The Art of Data Structures

Stacks



Alan Beadle
CSC 162: The Art of Data Structures

UNIVERSITY of ROCHESTER

Agenda

- What is a Stack?
- The Stack Abstract Data Type
- Implementing a Stack in Python
- String reversal
- Balanced Parentheses/symbols
- Infix, Prefix and Postfix Expressions

Linear Data Structures

Linear Data Structures

What are they?

- Data collections where items are ordered depending on how they are added/removed
- They stay in that order, relative to other elements before and after it

Linear Data Structures

What are they?

- Two ends (left, right, top, bottom, front, rear, etc...)
- Adding and removing is the distinguishing characteristic
  - May be limited on which end data is removed from or added to

Linear Data Structures

What are they?

- Simple, but powerful data structures will be covered
  - Stacks, Queues, Deques, and Lists
- Very useful in computer science
  - Appear in many algorithms, and solve important problems

Stacks

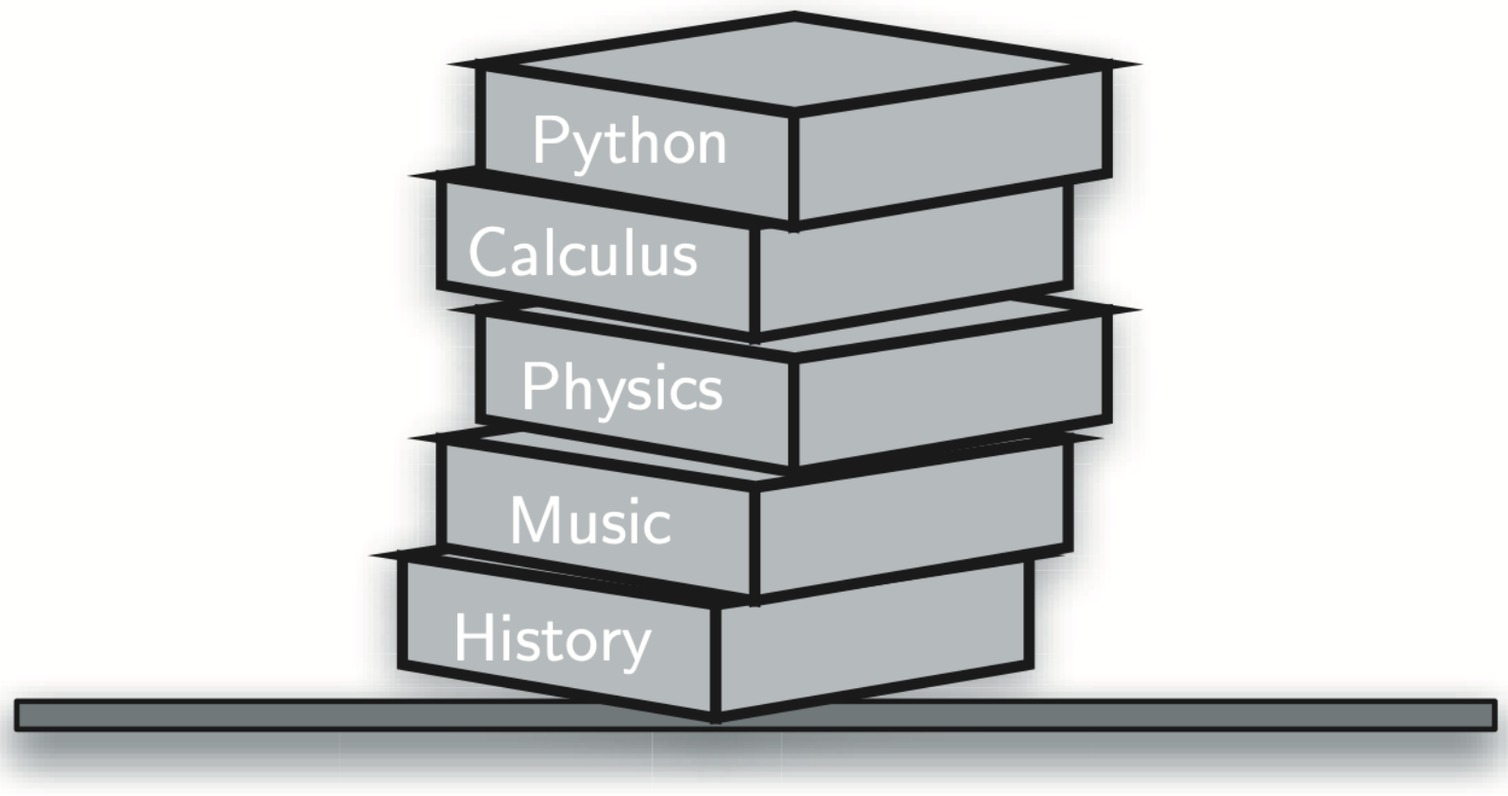
Stacks

Definition

- Also known as a "push-down stack"
- Ordered collection where items are added, or removed from the same end
- This means the last item added is the first one removed, also called a LIFO (last-in, first-out)
- The newest items are at the top/front and the oldest items are in the bottom/rear

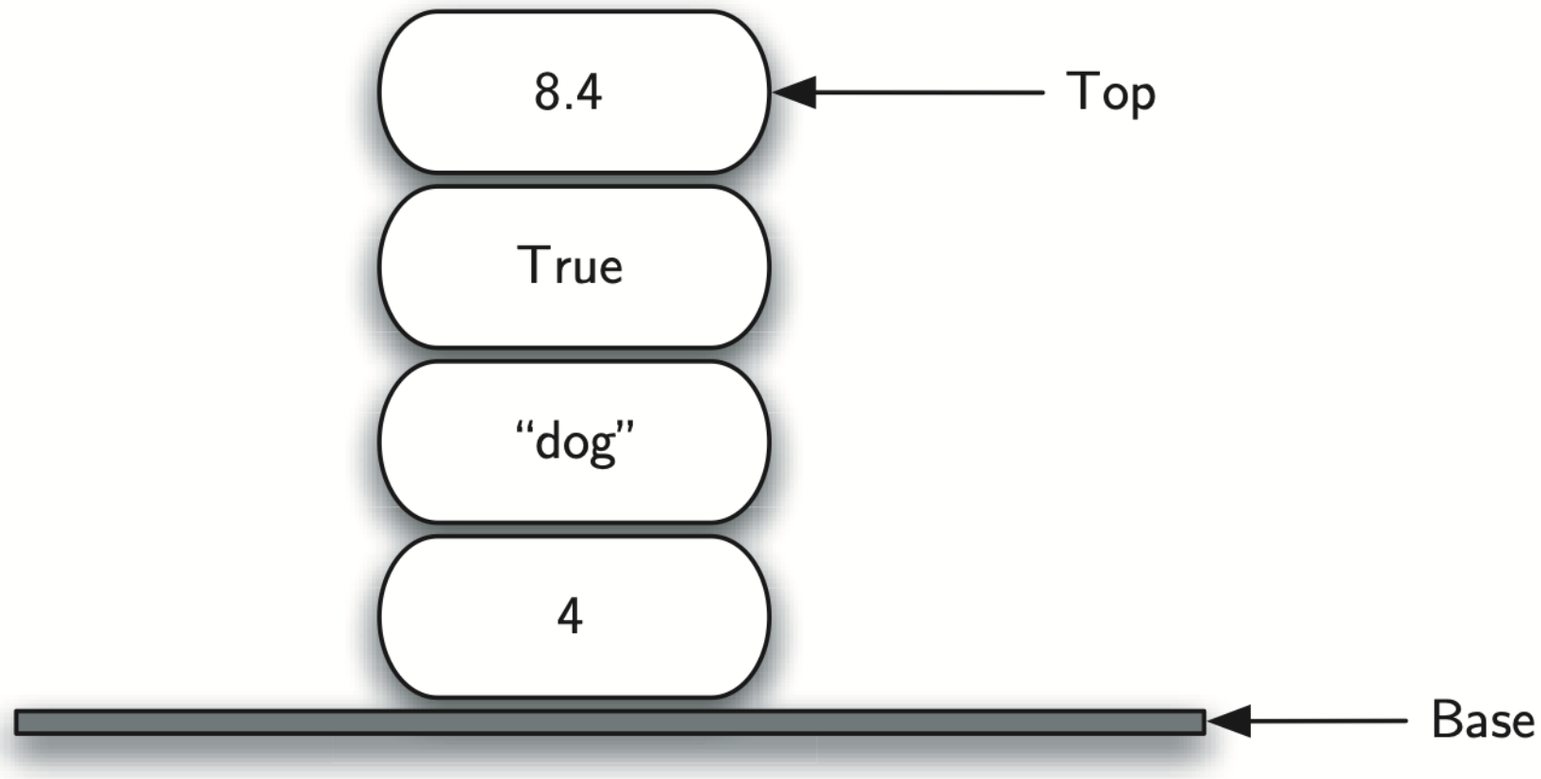
Stacks

A Stack of Books



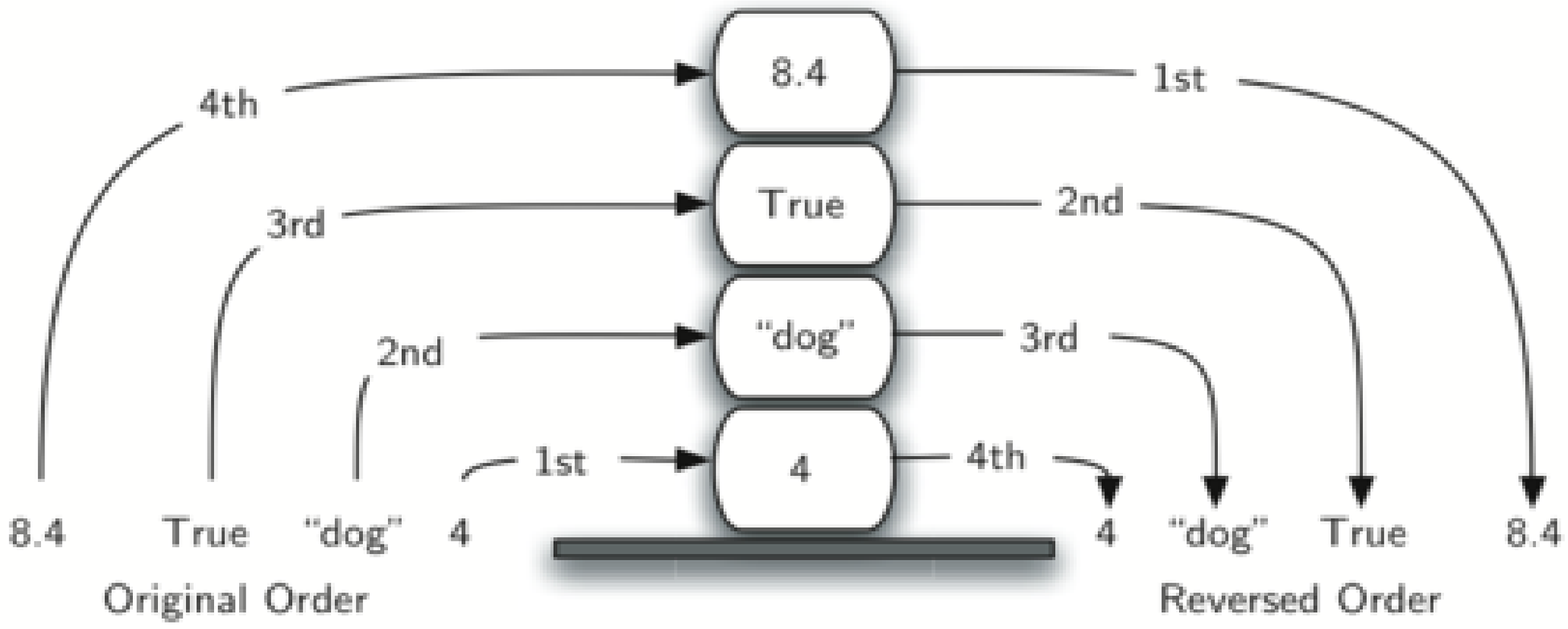
Stacks

A Stack of Primitive Python Objects



Stacks

The Reversal Property of Stacks



Implementation

Implementation
Stack Abstract Data Type

- The stack abstract data type is defined by:
  - The underlying data structure
  - The exposed actions that operate on it
- Items are removed, and added from the end called the "top"
  - They are an ordered LIFO

Implementation
Stack Abstract Data Type

- When an ADT is given a physical implementation, we refer to that implementation as a *data structure*
- We will be using Python classes to implement this ADT, where the stack operations will be methods
- A Python list will be the underlying data structure, as it is a powerful, and simply primitive collection structure

Implementation
Stack Operations

- `Stack()` creates a new stack that is empty; it needs no parameters and returns an empty stack
- `push(item)` adds a new item to the top of the stack; it needs the item and returns nothing
- `pop()` removes the top item from the stack; it needs no parameters, returns the item and the stack is modified

Implementation
Stack Operations

- `peek()` returns the top item from the stack but does not remove it; it needs no parameters; the stack is not modified
- `is_empty()` tests to see whether the stack is empty; it needs no parameters and returns a boolean value
- `size()` returns the number of items on the stack. It needs no parameters and returns an integer

Stack Operation	Stack Contents	Return Value
s.is_empty()	[]	True
s.push(4)	[4]	
s.push('dog')	[4, 'dog']	
s.peek()	[4, 'dog']	dog'
s.push(True)	[4, 'dog', True]	
s.size()	[4, 'dog', True]	3
s.is_empty()	[4, 'dog', True]	False
s.push(8.4)	[4, 'dog', True, 8.4]	
s.pop()	[4, 'dog', True]	8.4
s.pop()	[4, 'dog']	True
s.size()	[4, 'dog']	2

Implementation

Stack Implementation in Python

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.append(item)

    def pop(self):
        return self.items.pop()

    def peek(self):
        return self.items[-1]

    def size(self):
        return len(self.items)
```

Implementation
Alternate Implementation

- The *abstract* nature of an ADT can allow us to change the underlying physical implementation without affecting the logical characteristics
- The performance, however, may differ wildly
- The following alternate implementation will have push() and pop() methods that run in O(n), for a stack of size n

Implementation

Stack Implementation in Python

(Alternate, Slower)

```
class Stack:
    def __init__(self):
        self.items = []

    def is_empty(self):
        return self.items == []

    def push(self, item):
        self.items.insert(0, item)

    def pop(self):
        return self.items.pop(0)

    def peek(self):
        return self.items[0]

    def size(self):
        return len(self.items)
```

String reversal example

(In notebook)

Simple Balanced Parentheses

Simple Balanced Parentheses

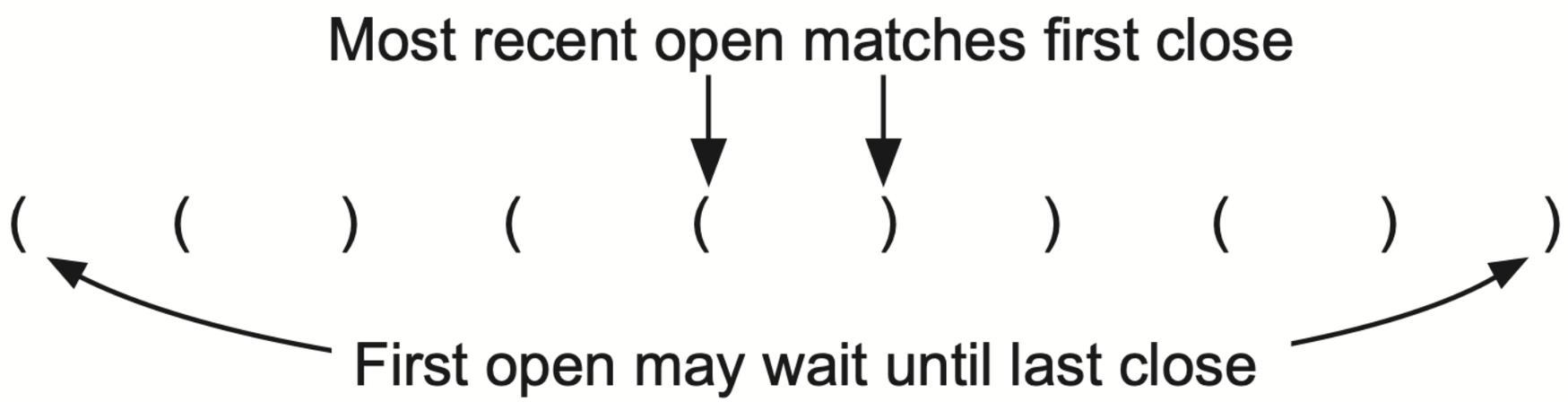
Matching Parentheses

(5+6)*(7+8)/(4+3)

LISP code example:

```
(defun square(n)
  (* n n))
```

“LISt Processing” or “Lost In Senseless Parentheses”?



Simple Balanced Parentheses

Parentheses Checker

```
def par_checker(symbol_string):
    s = Stack()
    balanced = True
    index = 0

    while index < len(symbol_string) and
balanced:
        symbol = symbol_string[index]
        if symbol == "(":
            s.push(symbol)
        else:
            # the case for a right/closing parens
            if s.is_empty():
                balanced = False
            else:
                s.pop()

        index += 1

    if balanced and s.is_empty():
        return True
    else:
        return False
```

Simple Balanced Parentheses
Parentheses Checker

- That could have been done more easily with just a counter

- If we extend to handle brackets and braces, too, we really need the stack...

Balanced Symbols (A General Case)

Balanced Symbols (A General Case)

```
def symbol_checker(symbol_string):
    s = Stack()
    balanced = True
    index = 0

    while index < len(symbol_string) and balanced:
    symbol = symbol_string[index]
    if symbol in '([{<':
        s.push(symbol)
    else:
        if s.is_empty():
            balanced = False
        else:
            top = s.pop()
            print(s.items, top, symbol)
            if not matches(top, symbol):
                balanced = False
    index += 1

return (balanced and s.is_empty())
```

Balanced Symbols (A General Case)

```
    def matches(_open, close):
        openers = "([{<"
        closers = ")]}>"
        return openers.index(_open) == closers.index(close)

print(par_checker('{{([][])}()}'))
print(par_checker('[{()]'))
```

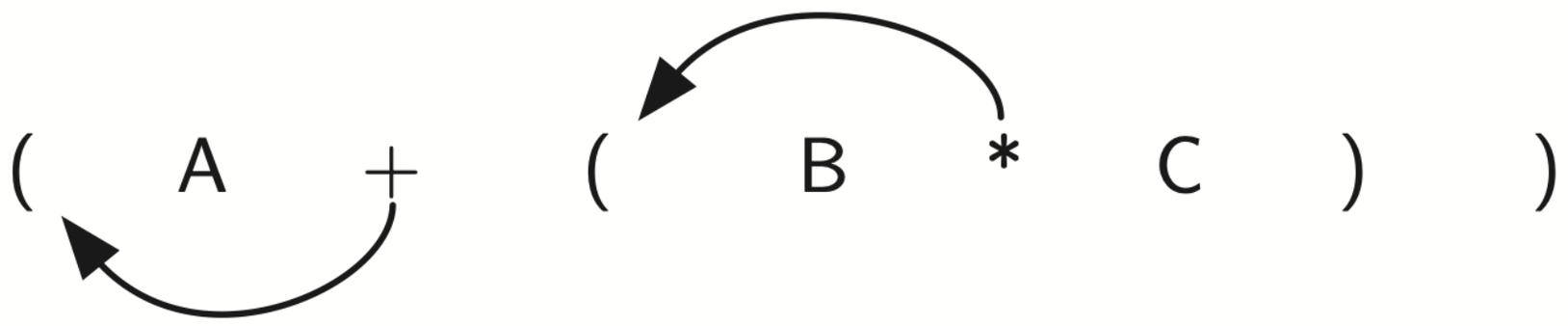
Infix, Prefix, Postfix Exps.

- This stuff can be confusing at first, so let’s start with a demonstration of postfix notation
- FORTH is a whole programming language based on stacks and postfix notation!
- As a hobby, I re-implemented an extinct dialect of FORTH called DSSP
- Let’s see how DSSP does math using a stack...

Infix, Prefix, Postfix Exps.

Moving Operators Rightward for Postfix Notation

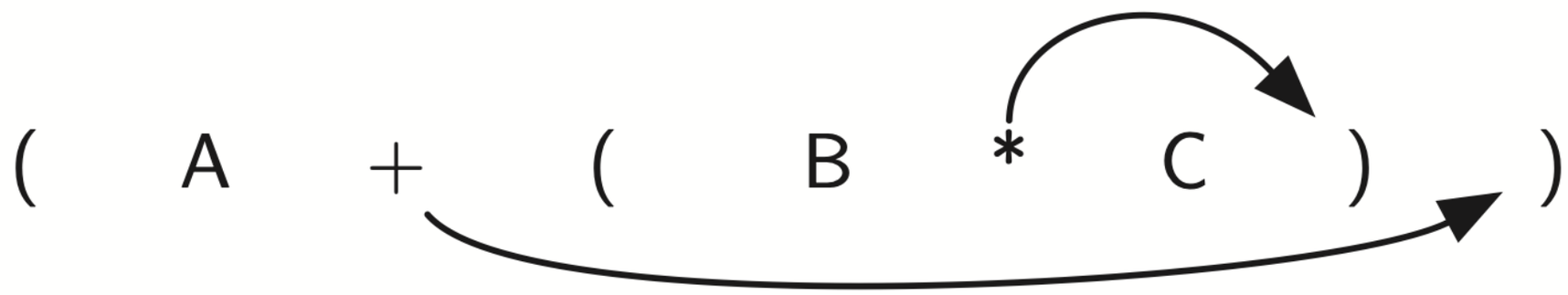
- We're used to infix: A + B
- We're also used to prefix, perhaps with parentheses:
- add(A, B)
- + A B



Infix, Prefix, Postfix Exps.

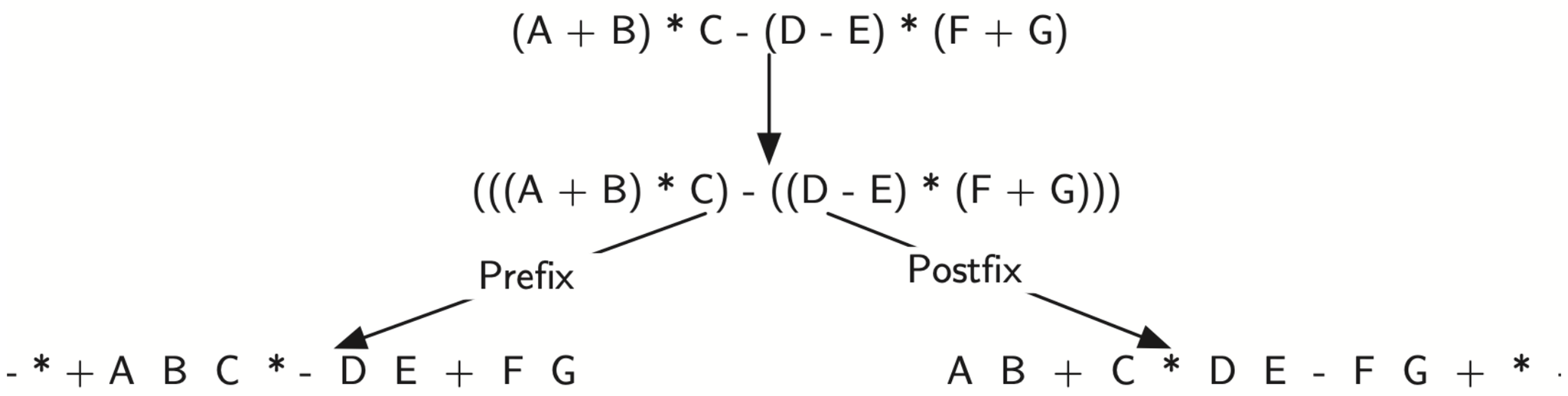
Moving Operators Leftward for Postfix Notation

- Postfix may seem a little strange: A B +



Infix, Prefix, Postfix Exps.

Converting a Complex Expression to Prefix and Postfix Notations



Infix, Prefix, Postfix Exps.

Converting a Complex Expression to Prefix and Postfix Notations

- With infix, we need parentheses to determine order of operations
- We sometimes leave them out if the order can be implied by rules for precedence:
  - 2 + 3 * 4 == 2 + (3 * 4) ==> 14
  - 2 + 3 * 4 == (2 + 3) * 4 ==> 20

Infix, Prefix, Postfix Exps.

Converting a Complex Expression to Prefix and Postfix Notations

- We sometimes leave them out if the order can be implied by rules for associativity:
  - 10 - 4 - 3 == (10 - 4) - 3 ==> 3
  - 10 - 4 - 3 == 10 - (4 - 3) ==> 9

Infix, Prefix, Postfix Exps.

Converting a Complex Expression to Prefix and Postfix Notations

- Prefix and postfix have no need for parentheses; the order is manifest
- M&R point out that this is because we've effectively placed the operator in the position of the left or right paren, so it implies the grouping

Infix, Prefix, Postfix Exps.

Converting a Complex Expression to Prefix and Postfix Notations

- `( A + ( B * C ) )` ← infix
- `+ A * B C` ← prefix
- `A B C * +` ← postfix

Infix, Prefix, Postfix Exps.

Converting a Complex Expression to Prefix and Postfix Notations

- `( ( A + B ) * C )` ← infix
- `* + A B C` ← prefix
- `A B + C *` ← postfix

Infix, Prefix, Postfix Exps.

Converting a Complex Expression to Prefix and Postfix Notations

- So what does this have to do with stacks?
- We can build on the balanced parens example to convert among the three representations and to evaluate expressions in any of the three notations
- Compilers and interpreters (like the Python interpreter) do a lot of this sort of thing

Infix, Prefix, Postfix Exps.

Converting a Complex Expression to Prefix and Postfix Notations

- We'll show to convert infix to postfix using a stack
  - One handles precedence, the other does not
  - Both have left associativity
  - Neither example handles right associativity (e.g., for exponentiation)

Infix, Prefix, Postfix Exps.

Infix to Postfix Notation (No Precedence)

```
import string

def infix_to_postfix(infix_expr):
    operand_stack = Stack()
    postfix_list = []
    token_list = infix_expr.split()

    for token in token_list:
        if token in string.ascii_uppercase:
            postfix_list.append(token)
        elif token == '(':
            operand_stack.push(token)
        elif token == ')':
            top_token = operand_stack.pop()
            while top_token != '(':
                postfix_list.append(top_token)
                top_token = operand_stack.pop()
        else: # Operator
            if not operand_stack.is_empty() and operand_stack.peek() != '(':
                postfix_list.append(operand_stack.pop())
            operand_stack.push(token)

    while not operand_stack.is_empty():
        postfix_list.append(operand_stack.pop())

    return "".join(postfix_list)
```

Infix, Prefix, Postfix Exps.

*Converting A * B + C * D to Postfix Notation*

1. The following algorithm handles precedence; still implicitly left associative
2. Create an empty stack called `opstack` for keeping operators
3. Create an empty list for output
4. Convert the input infix string to a list by using the string method `split`

Infix, Prefix, Postfix Exps.

*Converting A * B + C * D to Postfix Notation*

4. Scan the token list from left to right
   - If the token is an operand, append it to the end of the output list
   - If the token is a left parenthesis, push it on the `opstack`

Infix, Prefix, Postfix Exps.

*Converting A * B + C * D to Postfix Notation*

4. *(cont.)* Scan the token list from left to right
   - If the token is a right parenthesis, pop the `opstack` until the corresponding left parenthesis is removed
     - Append each operator to the end of the output list

Infix, Prefix, Postfix Exps.

*Converting A * B + C * D to Postfix Notation*

4. *(cont.)* Scan the token list from left to right
   - If the token is an operator, *, /, +, or -, push it on the `opstack`
   - However, first remove any operators already on the `opstack` that have higher or equal precedence and append them to the output list

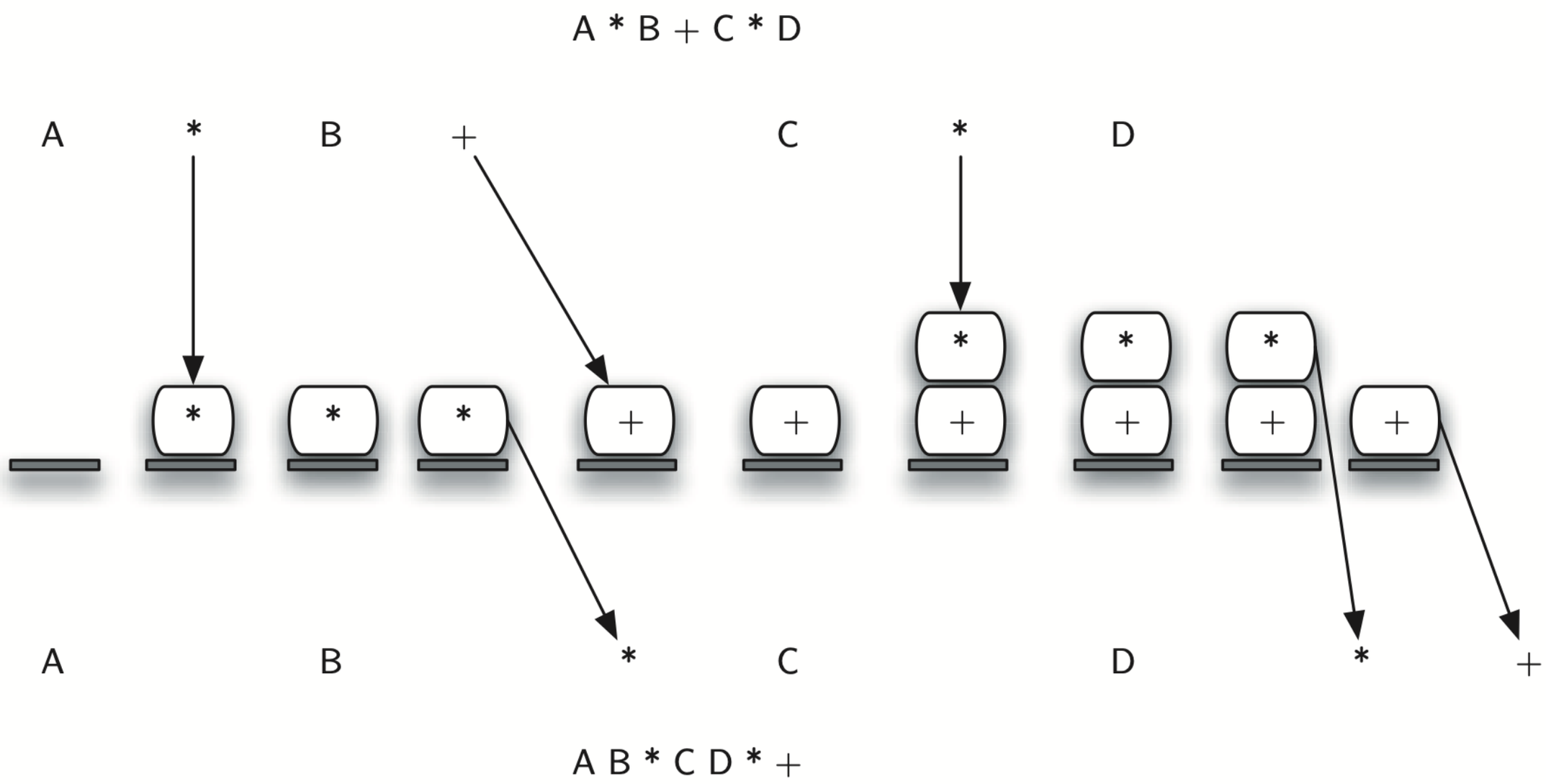
Infix, Prefix, Postfix Exps.

*Converting A * B + C * D to Postfix Notation*

5. When the input expression has been completely processed, check the `opstack`
   - Any operators still on the stack can be removed and appended to the end of the output list

Infix, Prefix, Postfix Exps.

*Converting A * B + C * D to Postfix Notation*



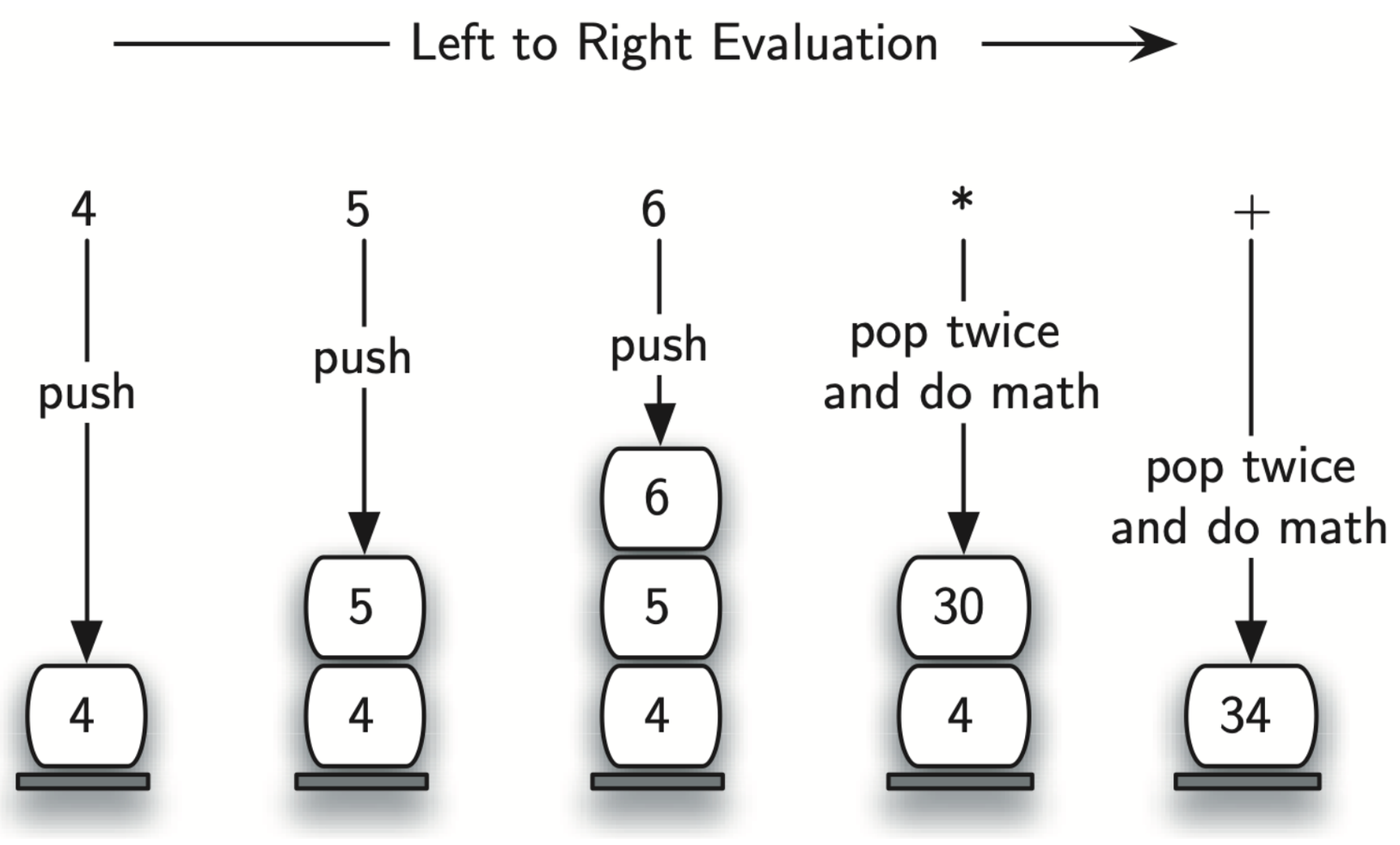
Infix, Prefix, Postfix Exps.

Infix to Postfix Notation (With Precedence)

```
def infix_to_postfix2(infix_expr):
    prec = {'*': 3, '/': 3, '+': 2, '-': 2, '(': 1}

    operand_stack = Stack()
    postfix_list = []
    token_list = infix_expr.split()  # our infix_expr should have spaces in it!

    for token in token_list:
        if token in string.ascii_uppercase or token in string.digits:
            postfix_list.append(token)
        elif token == '(':
            operand_stack.push(token)
        elif token == ')':
            top_token = operand_stack.pop()
            while top_token != '(':
                postfix_list.append(top_token)
                top_token = operand_stack.pop()
        else:
            while (not operand_stack.is_empty() and
                   prec[operand_stack.peek()] >= prec[token]):
                postfix_list.append(operand_stack.pop())

            operand_stack.push(token)

    while not operand_stack.is_empty():
        postfix_list.append(operand_stack.pop())

    return " ".join(postfix_list)
```

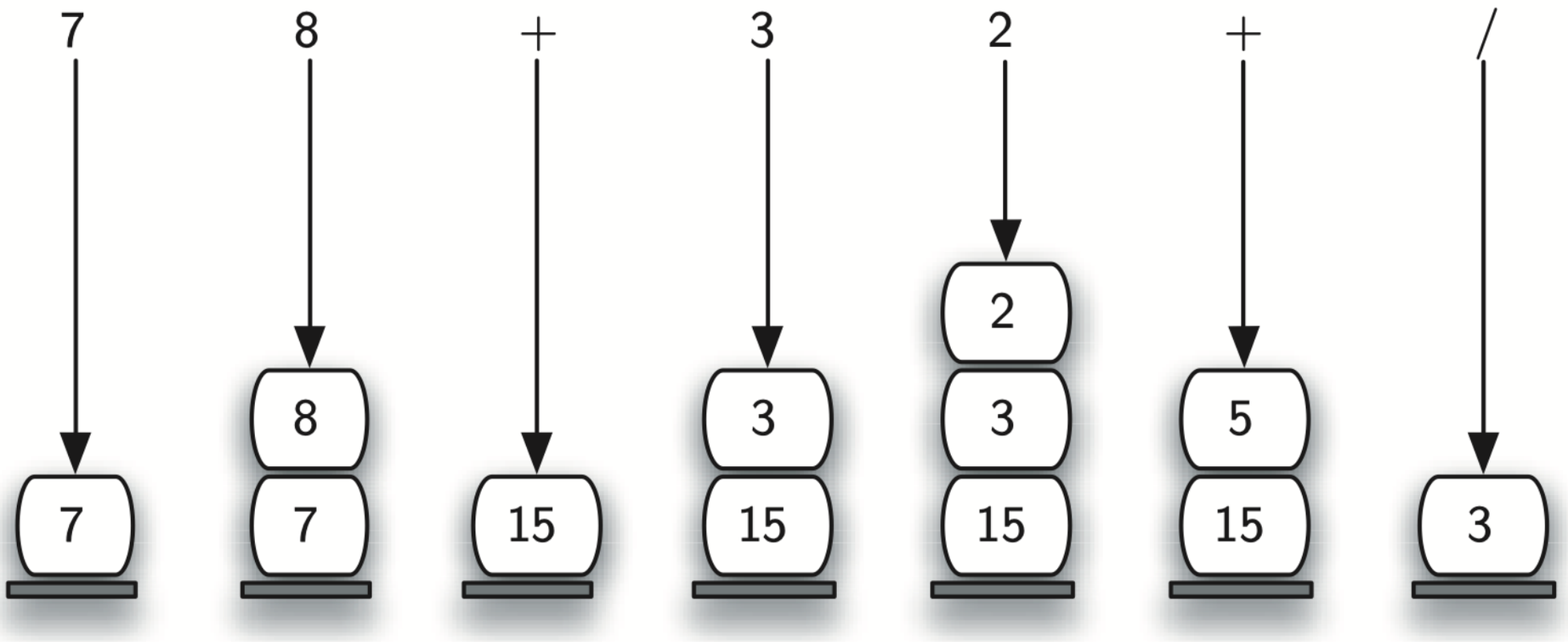
Infix, Prefix, Postfix Exps.

Stack Contents During Evaluation



Infix, Prefix, Postfix Exps.

A More Complex Example of Evaluation



Infix, Prefix, Postfix Exps.

Now let's write some Python code to evaluate postfix expressions!

1. Create an empty stack called `operand_stack`
2. Convert the string to a list by using the string method `split`

Infix, Prefix, Postfix Exps.

A More Complex Example of Evaluation

3. Scan the token list from left to right
   - If the token is an operand, convert it from a `str` to an `int`, then push onto the `operand_stack`
   - If the token is an operator, *, /, +, or -, it will need two operands
   - Pop the `operand_stack` twice
   - The first pop is the second operand and the second pop is the first operand

Infix, Prefix, Postfix Exps.

A More Complex Example of Evaluation

4. Scan the token list from left to right *(cont.)*

   - The first pop is the second operand and the second pop is the first operand
   - Perform the arithmetic operation
   - Push the result back on the `operand_stack`

Infix, Prefix, Postfix Exps.

Postfix Evaluation

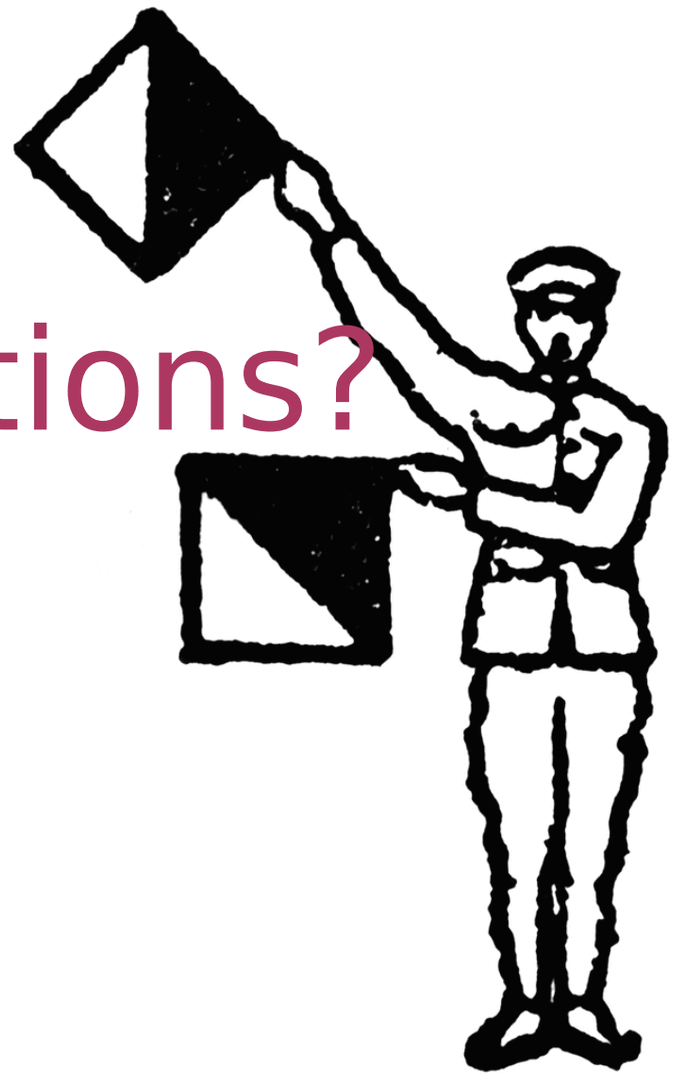
```
def postfix_eval(postfix_expr):
    operand_stack = Stack()  # operAND stack
    token_list = postfix_expr.split()

    for token in token_list:
        if token in string.digits:
            operand_stack.push(int(token))
        else:
            operand2 = operand_stack.pop()
            operand1 = operand_stack.pop()
            result = do_math(token, operand1, operand2)
            # Can you eliminate `do_math` with a one-liner?
            operand_stack.push(result)

    return operand_stack.pop()
```

Infix, Prefix, Postfix Exps.

Postfix Evaluation

```
def do_math(op, op1, op2):
    if op == '*':
        return op1 * op2
    elif op == '/':
        return op1 / op2
    elif op == '+':
        return op1 + op2
    else:
        return op1 - op2

print(postfixEval('7 8 + 3 2 + /'))
```

Questions?



- Next, a workshop where you will add some methods to our Stack class

- A new lab assignment will appear at 11:55, due in 1 week

- Recall that your first lab is due in 2 days, and submissions for Friday's workshop close tonight!

- Email me anytime for advice or to schedule a Zoom meeting