CS46K Programming Languages

Structure and Interpretation of Computer Programs

Chapter 6B Functional Programming – Recursion and Tree

LECTURE 8: RECURSION AND TREE

DR. ERIC CHOU

IEEE SENIOR MEMBER

Objectives

- Mapping Functional programming onto parallel processors.
- Recursion
- Tail Recursion
- Tree and Graph
- Recursive Search
- Tree Evaluation
- Graph Evaluation
- Game Tree

The Beauty of Functional Programming

SECTION 1

What makes functional programming a viable choice for artificial intelligence projects?

- The most common programming languages currently used for AI and machine learning development are **Python, R, Scala, Go**, among others with the latest addition being **Julia**. Functional languages as old as **Lisp** and **Haskell** were used to implement machine learning algorithms decades ago when AI was an obscure research area of interest.
- There wasn’t enough hardware and software advancements back them for implementations. Some commonalities in all of the above language options are that they are simple to understand and promote clarity. They use fewer lines of code and lend themselves well to the functional programming paradigm.

Functional programming features

- Before we see functional programming in a machine learning context, let’s look at some of its characteristics.

Functional programming features

- **Immutable:** If a variable x is declared and used in the program, the value of the variable is never changed later anywhere in the program. Each time the variable x is called, it will return the same value assigned originally. This makes it pretty straightforward, eliminating the need to think of state change throughout the program.
- **Referential transparency:** This means that an expression or computation always results in the same value in any part/context of the program. A referentially transparent programming language's programs can be manipulated as algebraic equations.

Functional programming features

- **Lazy evaluation:** Being referentially transparent, the computations yield the same result irrespective of when they are performed. This enables to postpone the computation of values until they are required/called. This means one could evaluate them lazily. Lazy evaluation helps avoids unnecessary computations and saves memory.
- **Parallel programming:** Since there is no state change due to immutable variables, the functions in a functional program can work in parallel as instructions. Parallel loops can be easily expressed with good reusability.

Functional programming features

- **Higher-order functions:** A higher order function can take one or more functions as arguments. They may also be able to return a function as their result. Higher-order functions are useful for **refactoring code** and to **reduce repetition**. The map function found in many programming languages is an example of a higher-order function.

What kind of programming is good for AI development?

- Machine learning is a sub-domain of artificial intelligence which deals with concepts of making predictions from data, take actions without being explicitly programmed, recommendation systems and so on. Any programming approach that focuses on logic and mathematical functions is good for artificial intelligence (AI).
- Once the data is collected and prepared it is time to build your machine learning model.. This typically entails choosing a model, then training and testing the model with the data. Once the desired accuracy/results are achieved, then the model is deployed. Training on the data requires data to be consistent and the code to be able to communicate directly with the data without much abstraction for least unexpected errors.

What kind of programming is good for AI development?

- For AI programs to work well, the language needs to have a low level implementation for faster communication with the processor. This is why many machine learning libraries are created in C++ to achieve fast performance. **OOP with its mutable objects and object creation is better suited for high-level production software development, not very useful in AI programs which works with algorithms and data.**
- As AI is heavily based on math, languages like Python and R are widely used languages in AI currently. R lies more towards statistical data analysis but does support machine learning and neural network packages. Python being faster for mathematical computations and with support for numerical packages is used more commonly in machine learning and artificial intelligence.

Why is functional programming good for artificial intelligence?

- There are some benefits of functional programming that make it suitable for AI. It is closely aligned to mathematical thinking, and the expressions are in a format close to mathematical definitions. There are few or no side-effects of using a functional approach to coding, one function does not influence the other unless explicitly passed. This proves to be great for concurrency, parallelization and even debugging.

Less code and more consistency

- The functional approach uses fewer lines of code, without sacrificing clarity. More time is spent in thinking out the functions than actually writing the lines of code. But the end result is more productivity with the created functions and easier maintenance since there are fewer lines of code. AI programs consist of lots of matrix multiplications. Functional programming is good at this just like GPUs.
- You work with datasets in AI with some algorithms to make changes in the data to get modified data. A function on a value to get a new value is similar to what functional programming does. It is important for the variables/data to remain the same when working through a program. Different algorithms may need to be run on the same data and the values need to be the same. Immutability is well-suited for that kind of job.

Simple approach, fast computations

- The characteristics/features of functional programming make it a good approach to be used in artificial intelligence applications. AI can do without objects and classes of an object-oriented programming (OOP) approach, it needs fast computations and expects the variables to be the same after computations so that the operations made on the data set are consistent.
- Some of the popular functional programming languages are R, Lisp, and Haskell. The latter two are pretty old languages and are not used very commonly. Python can be used as both, functional and object oriented. Currently, Python is the language most commonly used for AI and machine learning because of its simplicity and available libraries. Especially the scikit-learn library provides support for a lot of AI-related projects.

FP is fault tolerant and important for AI

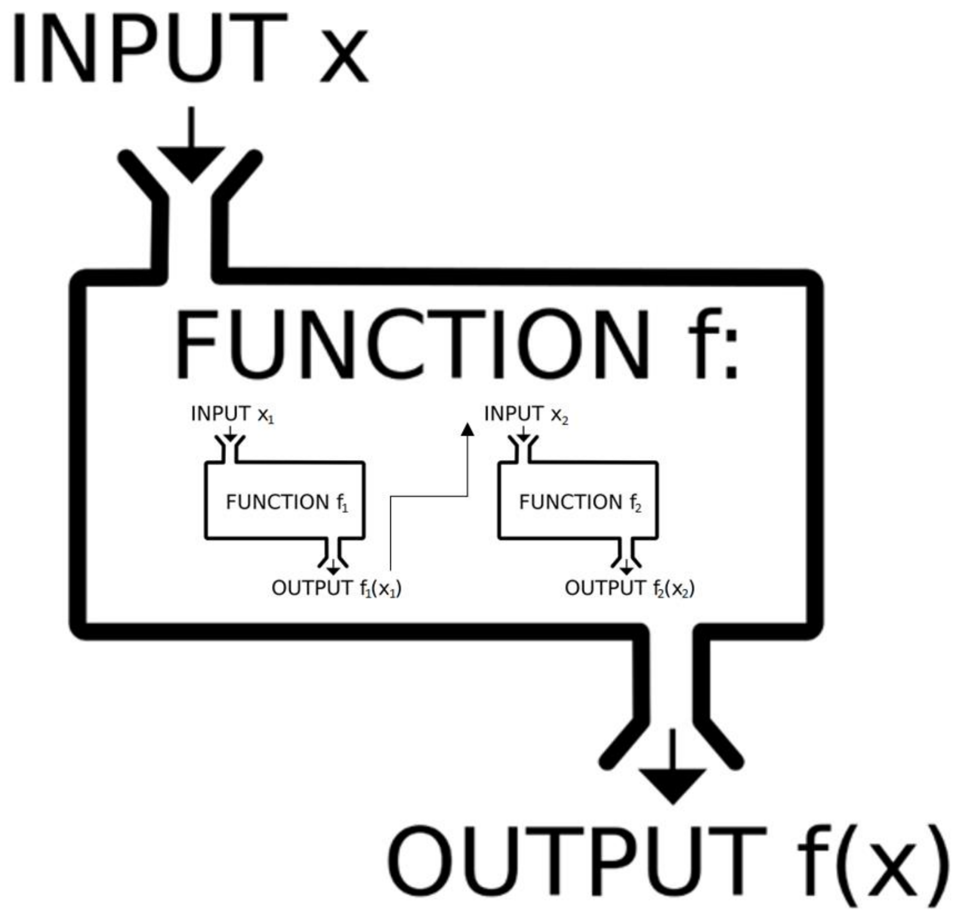
- Functional programming features make programs fault tolerant and fast for critical computations and rapid decision making. As of now, there may not be many such applications but think of the future, systems for self-driving cars, security, and defense systems. Any fault in such systems would have serious effects. Immutability makes the system more reliable, lazy evaluation helps conserve memory, parallel programming makes the system faster. The ability to pass a function as an argument saves a lot of time and enables more functionality. These features of functional programming make it a fitting choice for artificial intelligence.
- To further understand why use functional programming for machine learning, read the case made for using the functional programming language Haskell for AI in the Haskell Blog.

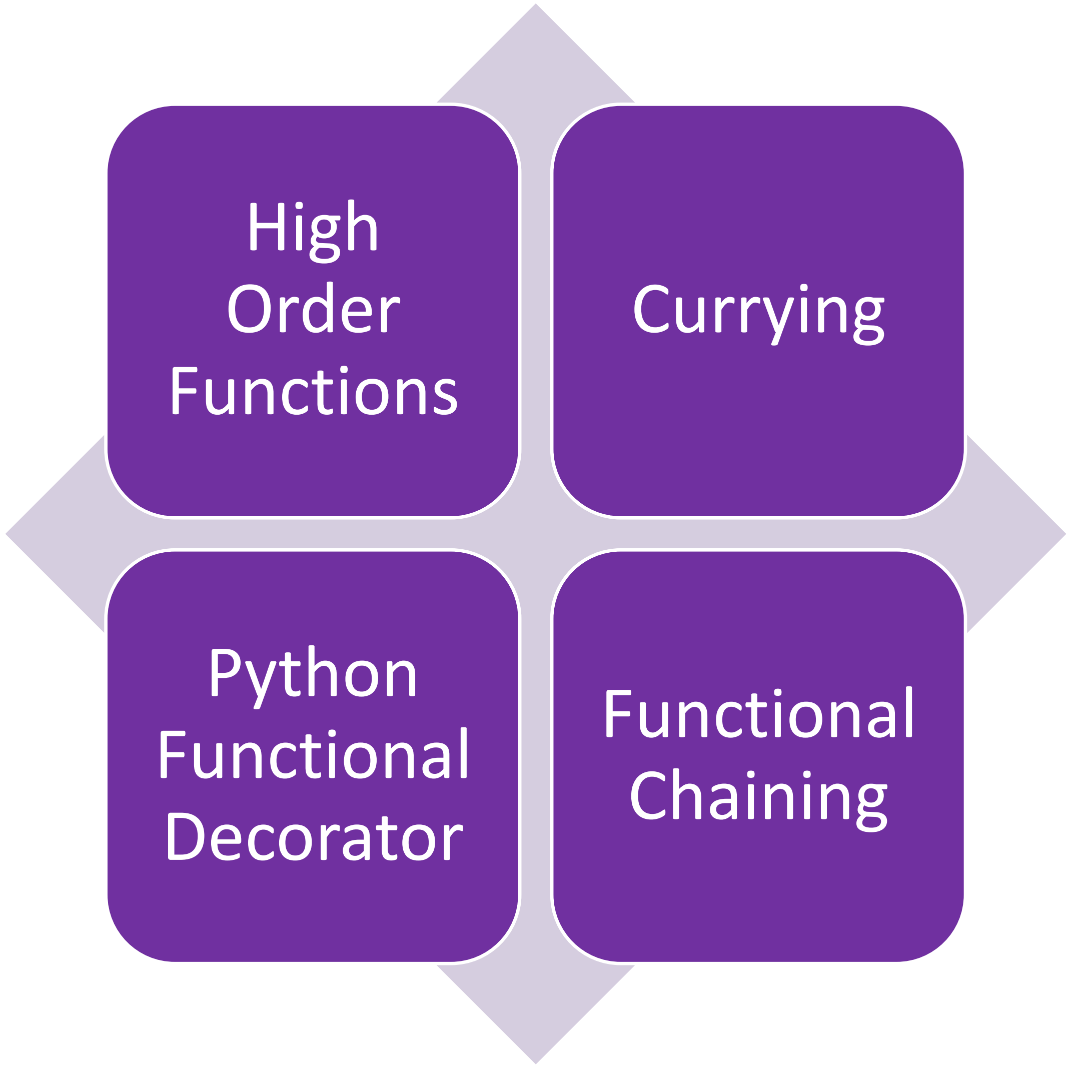
Powerful Programming Languages For Doing

Machine Learning

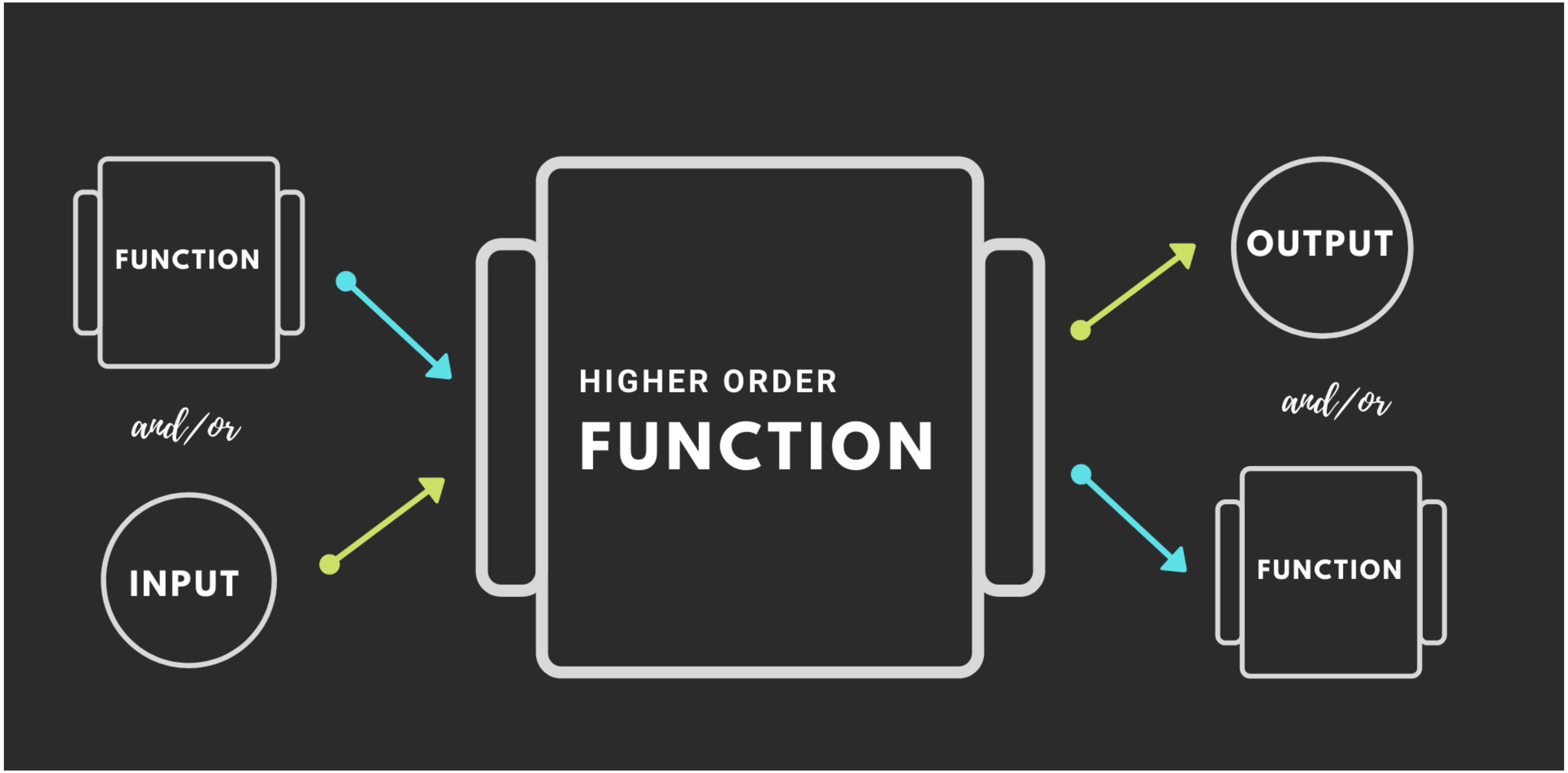
Program Struct for Functional Programming

SECTION 2


INPUT x
FUNCTION f:
INPUT x_1
FUNCTION f_1
OUTPUT $f_1(x_1)$
INPUT x_2
FUNCTION f_2
OUTPUT $f_2(x_2)$
OUTPUT f(x)

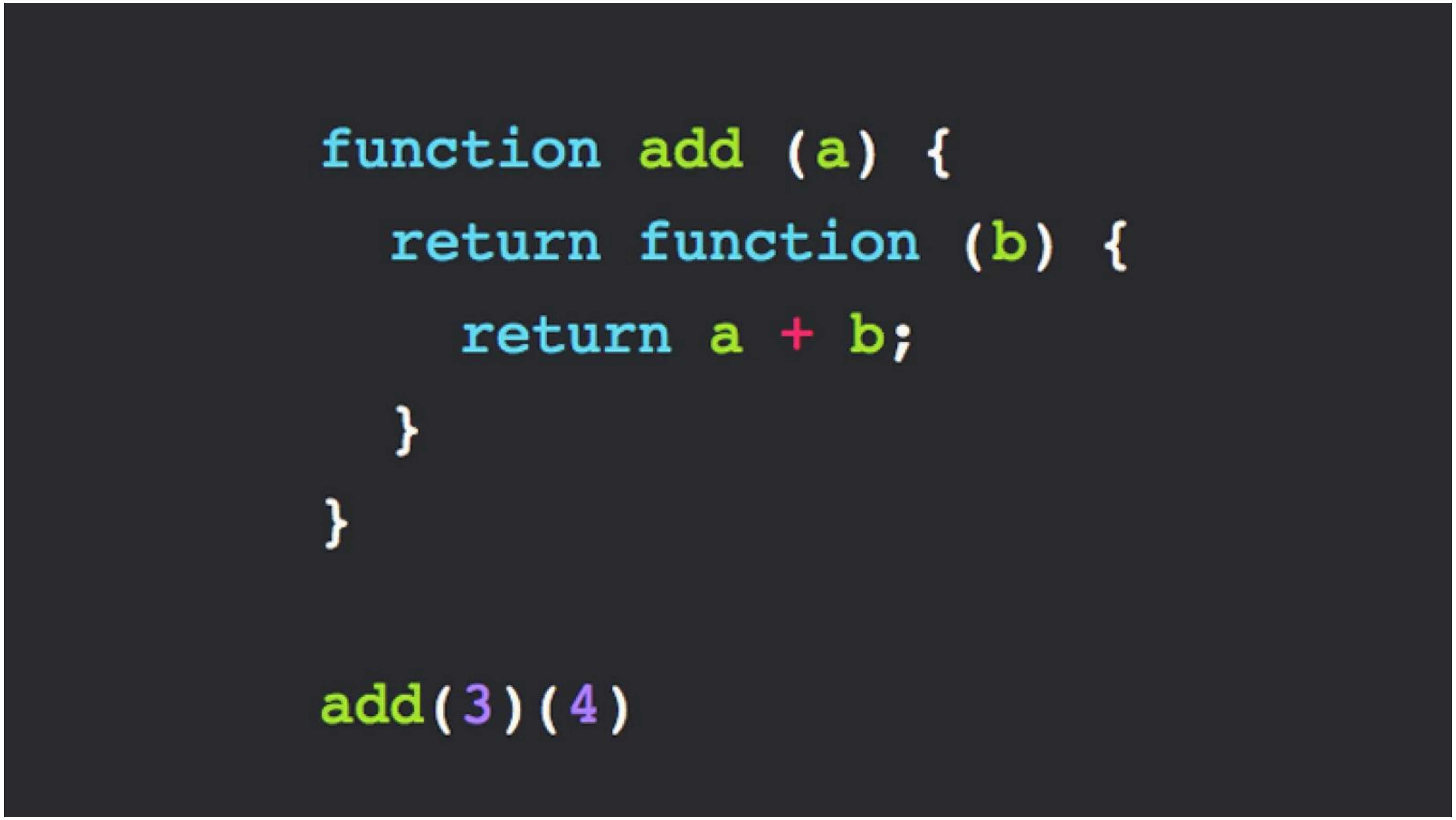


Program Struct for Functional Programming

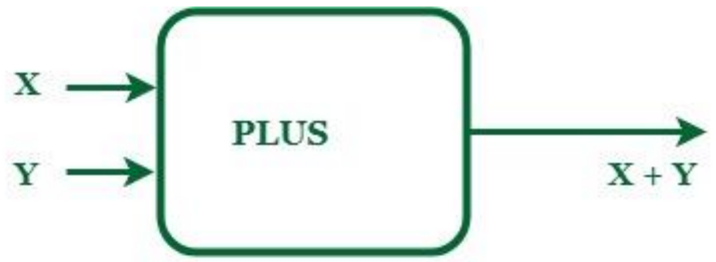

FUNCTION
and/or
INPUT
HIGHER ORDER
FUNCTION
OUTPUT
and/or
FUNCTION

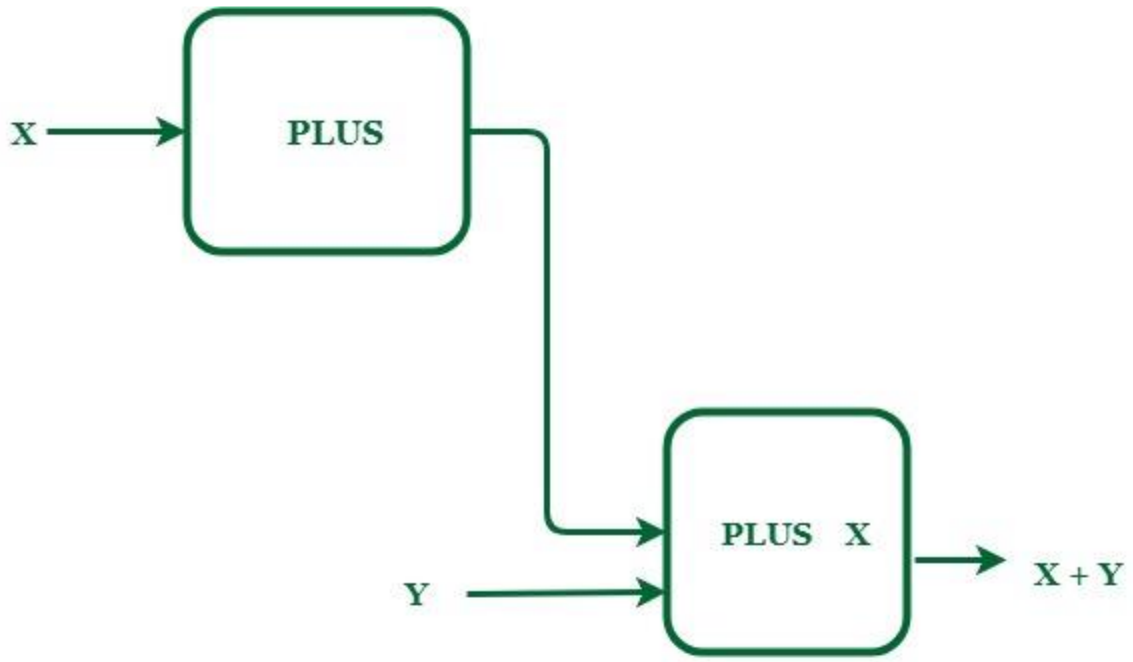
Out In	Vector	Function
Vector	Regular function	Function factory
Function	Functional	Function operator

Currying: Partial Argument Application with Functions in Python



```
function add (a) {
  return function (b) {
    return a + b;
  }
}

add(3)(4)
```





Normal Implementon of Function

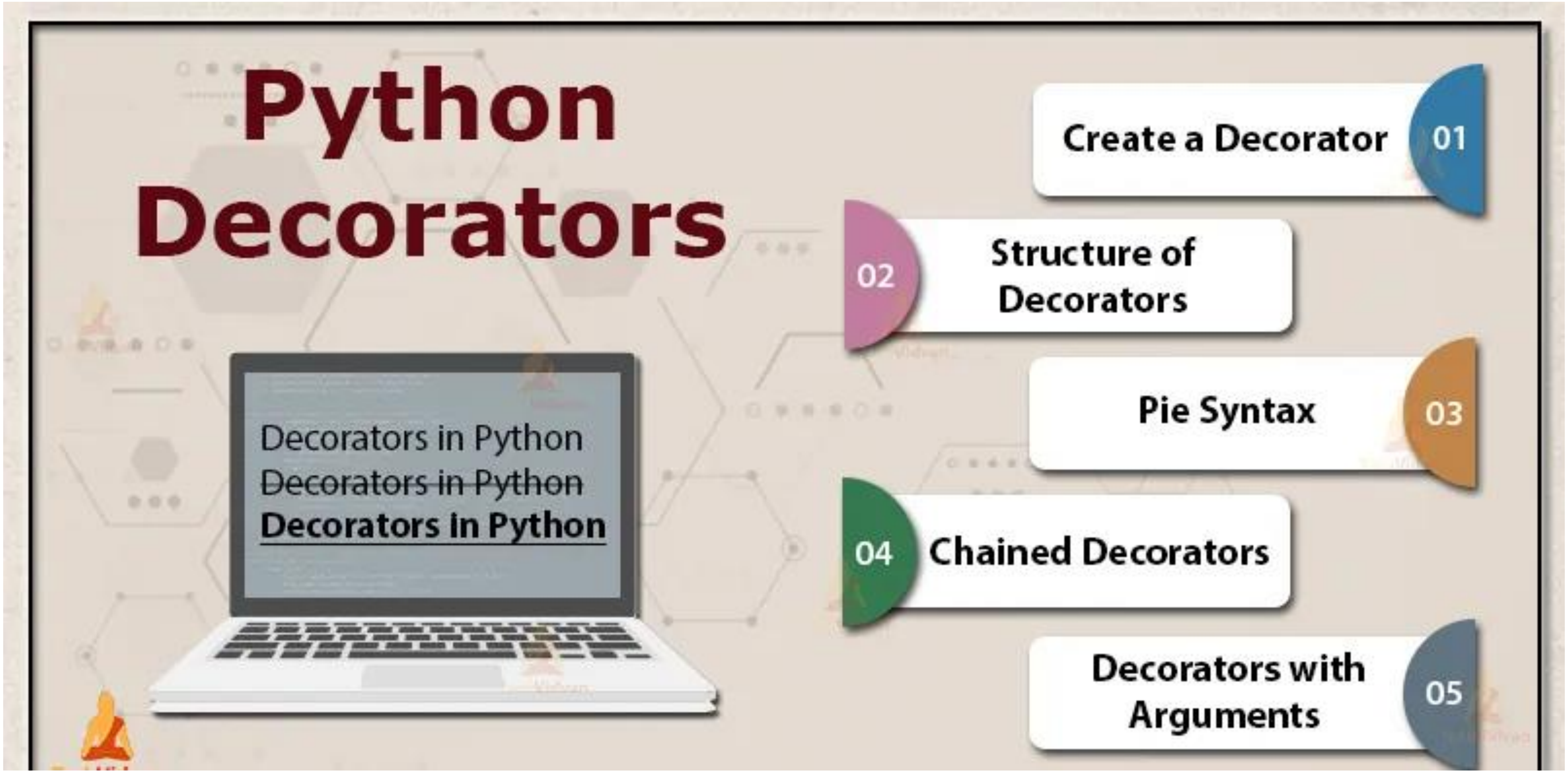
Implemention of Curried Function

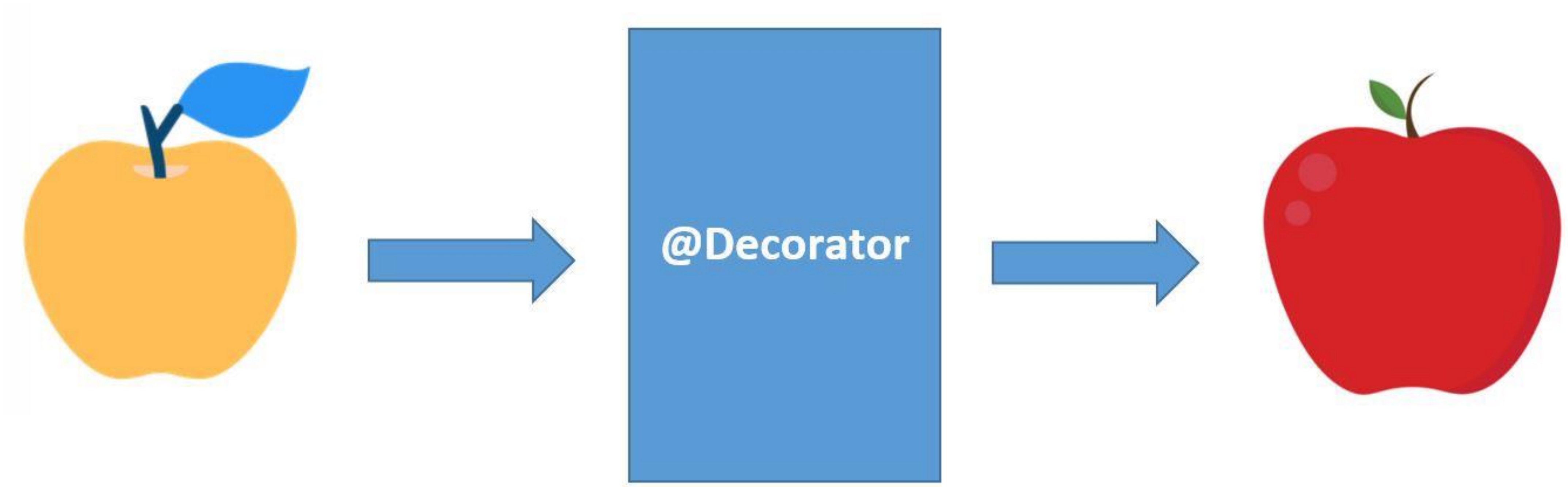
f: A × B × C -> D *non curried.*

f: A -> B -> C -> D *curried.*

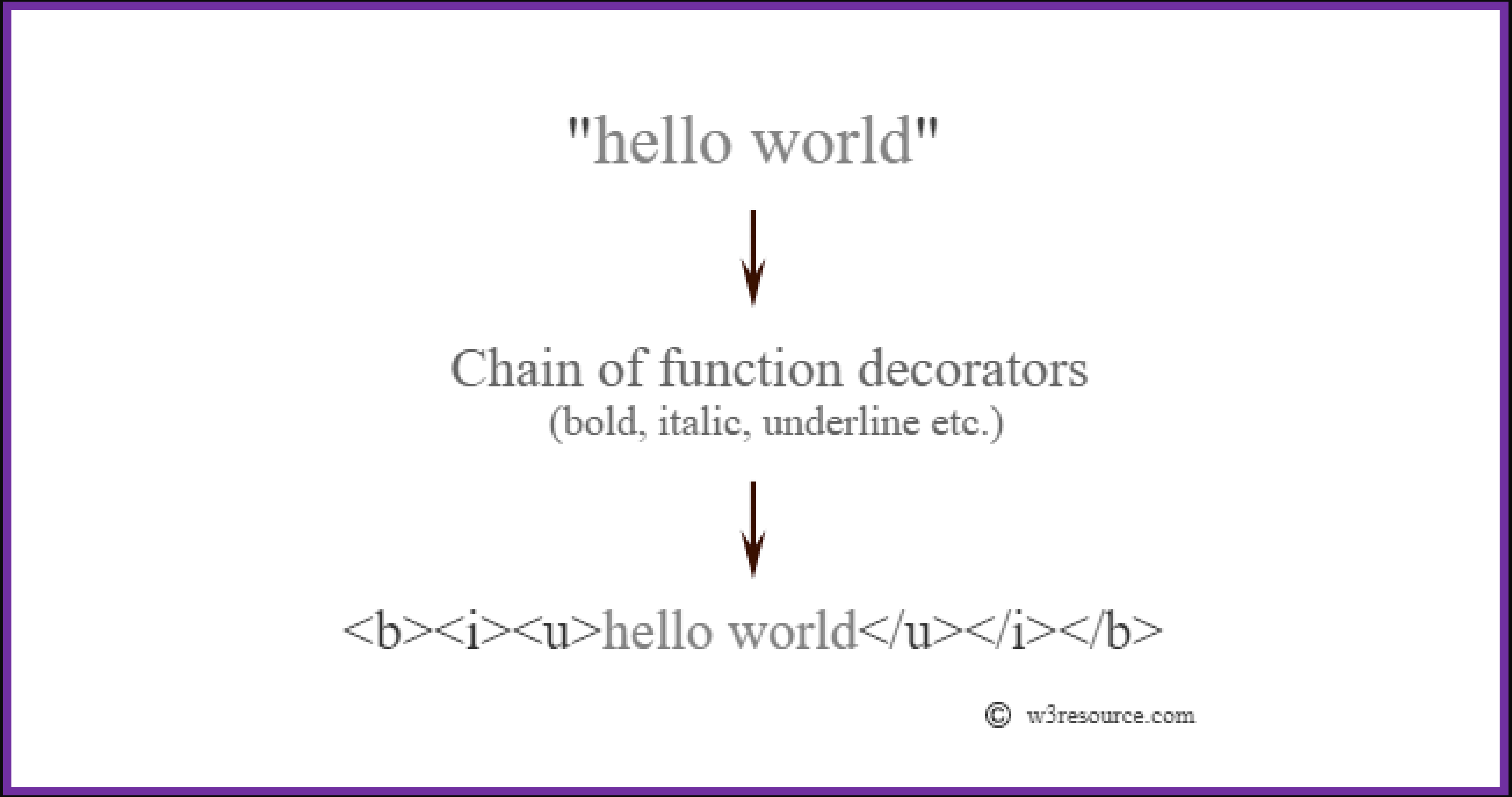
f:	A -> (B -> (C -> D))
f a:	B -> (C -> D)
f a b:	C -> D

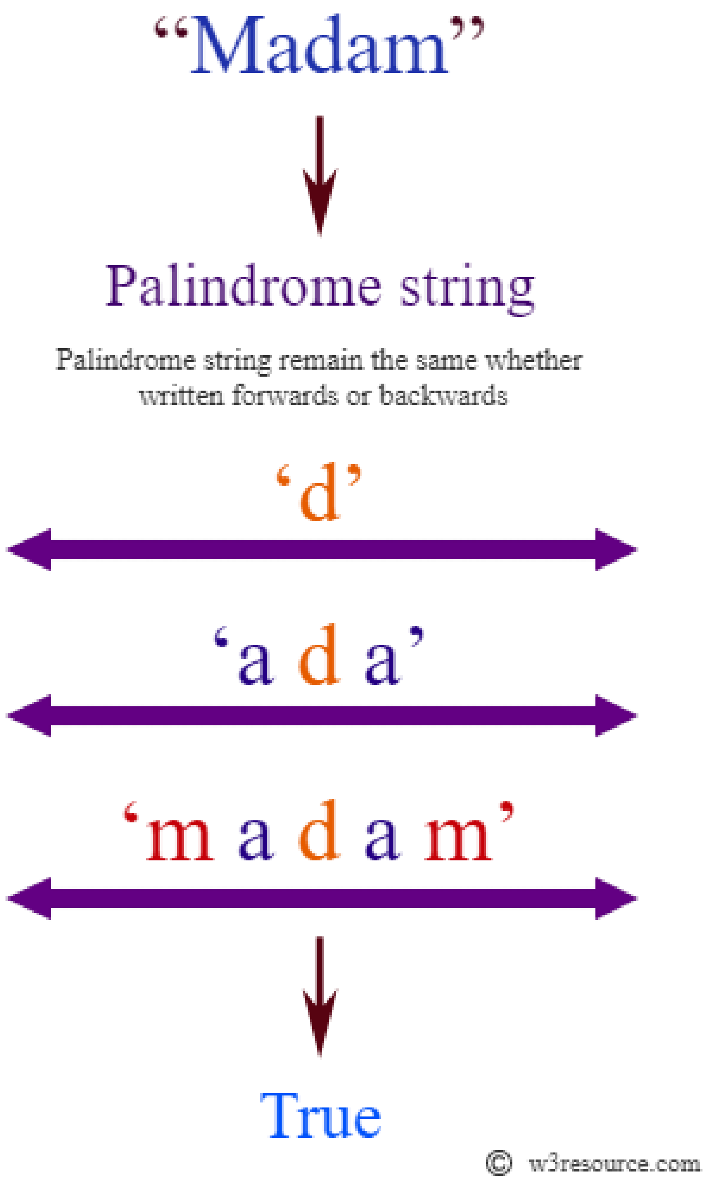
f:	A -> B -> C -> D
f a:	B -> C -> D
f a b:	C -> D

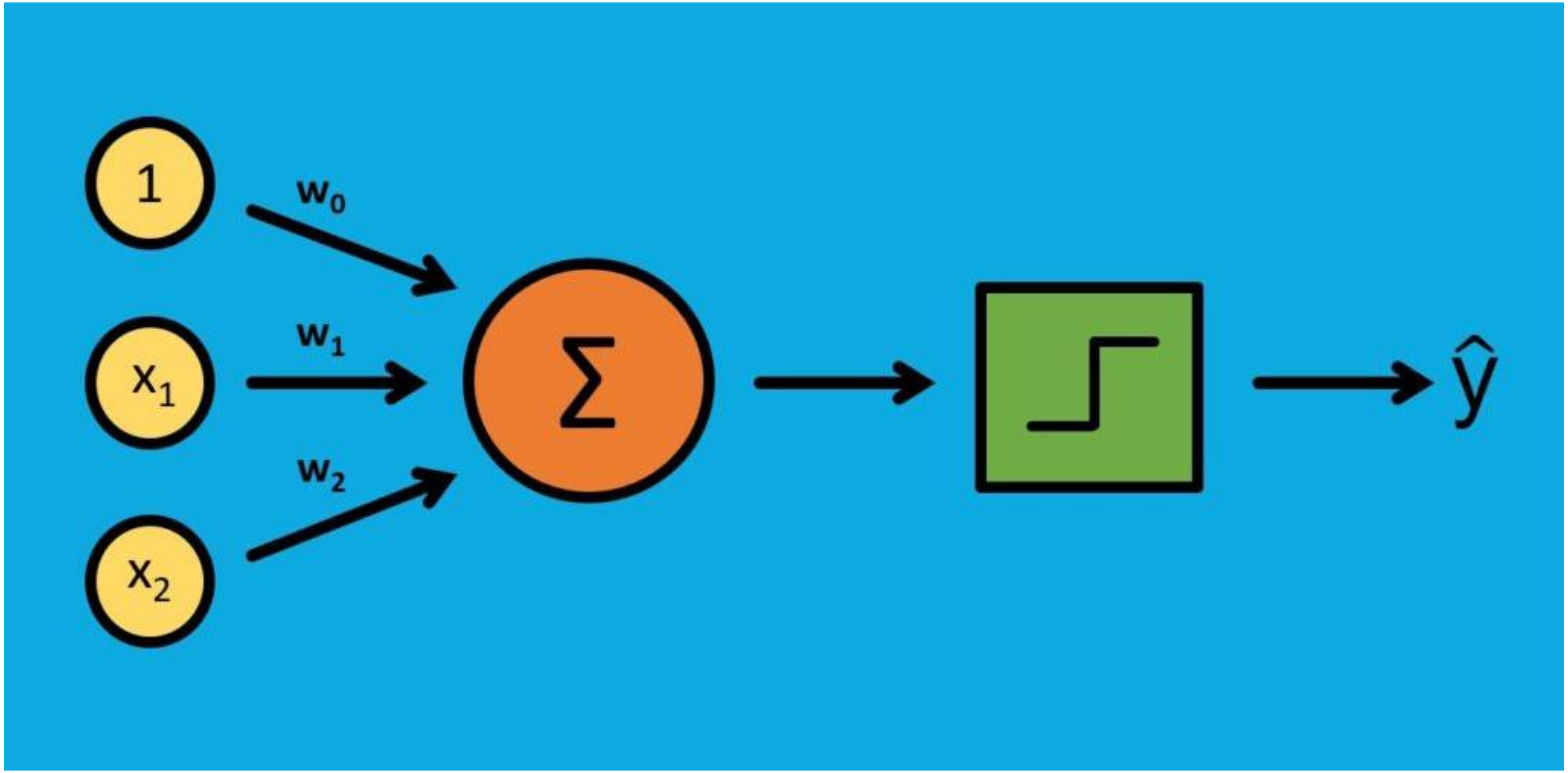

Python Decorators
Decorators in Python
~~Decorators in Python~~
Decorators in Python
Create a Decorator 01
02 Structure of Decorators
Pie Syntax 03
04 Chained Decorators
Decorators with Arguments 05

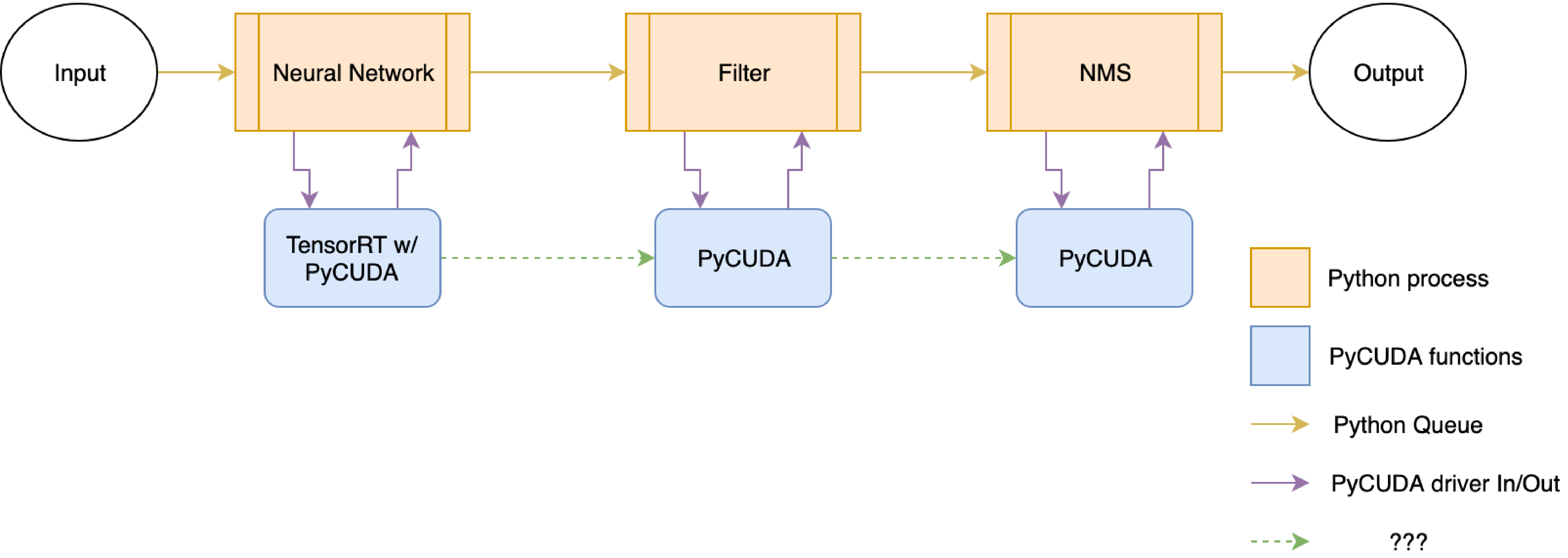

@Decorator

Python
Method Chaining
1
class [object]
method def A
method def B
ANACONDA


"hello world"
Chain of function decorators
(bold, italic, underline etc.)
<b><i><u>hello world</u></i></b>
© w3resource.com


“Madam”
Palindrome string
Palindrome string remain the same whether written forwards or backwards
‘d’
‘a d a’
‘m a d a m’
True
© w3resource.com


1
x_1
x_2
w_0
w_1
w_2
Σ
$\hat{y}$


Input
Neural Network
Filter
NMS
Output
TensorRT w/ PyCUDA
PyCUDA
PyCUDA
Python process
PyCUDA functions
Python Queue
PyCUDA driver In/Out
???

Python Higher Order Functions

SECTION 3

Higher Order Functions in Python

- A function is called **Higher Order Function** if it contains other functions as a parameter or returns a function as an output i.e, the functions that operate with another function are known as **Higher order Functions**.
- It is worth knowing that this higher order function is applicable for functions and methods as well that takes functions as a parameter or returns a function as a result. Python too supports the concepts of higher order functions.

Properties of higher-order functions

- A function is an instance of the Object type.
- You can store the function in a variable.
- You can pass the function as a parameter to another function.
- You can return the function from a function.
- You can store them in data structures such as hash tables, lists, …

Functions as objects

- In Python, a function can be assigned to a variable. This assignment does not call the function, instead a reference to that function is created. Consider the below example, for better understanding.

Function as Objects

high1.py

```
# Python program to illustrate functions
# can be treated as objects
def shout(text):
    return text.upper()

print(shout('Hello'))

# Assigning function to a variable
yell = shout

print(yell('Hello’))
```

```
HELLO
HELLO
```

Passing Function as an argument to other function

- Functions are like objects in Python, therefore, they can be passed as argument to other functions. Consider the below example, where we have created a function greet which takes a function as an argument.

Passing Function as an argument to other function

high2.py

```
def shout(text):
    return text.upper()
def whisper(text):
    return text.lower()
def greet(func):
    # storing the function in a variable
    greeting = func("Hi, I am created by a function \
    passed as an argument.")
    print(greeting)

greet(shout)
greet(whisper)
```

```
HI, I AM CREATED BY A FUNCTION    PASSED AS AN ARGUMENT.
hi, i am created by a function    passed as an argument.
```

Returning function

- As functions are objects, we can also return a function from another function. In the below example, the create_adder function returns adder function.

high3.py

```
def create_adder(x):
    def adder(y):
        return x + y
    return adder

add_15 = create_adder(15)

print(add_15(10))
```

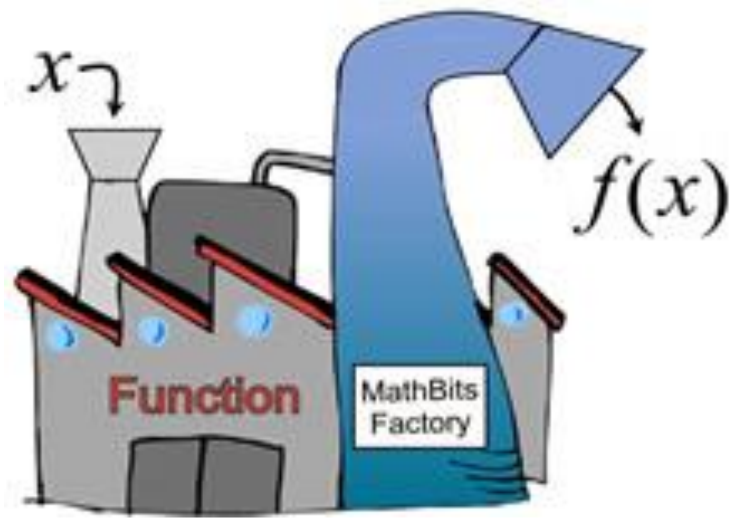
```
25
```

Currying

SECTION 4

Function currying

- **Currying:** Converting a function that takes multiple arguments into a single-argument higher-order function.
- A function that currys any two-argument function:



Function currying

- A function that currys any two-argument function:

curry2.py

```
def curry2(f):
    def g(x):
        def h(y):
            return f(x, y)
        return h
    return g

from operator import add
make_adder = curry2(add)
x = make_adder(2)(3)
print(x)
```

```
5
```

Function currying

- A function that currys any two-argument function:

curry1.py

```
curry2 = lambda f: lambda x: lambda y: f(x, y)

from operator import add
make_adder = curry2(add)
x = make_adder(2)(3)
print(x)
```

5

Use case for currying #1

- Whenever another function requires a function that only takes one argument:

curry3.py

```
from operator import add
curry2 = lambda f: lambda x: lambda y: f(x, y)

def transform_numbers(num1, num2, num3, transform):
    return (transform(num1), transform(num2), transform(num3))

x = transform_numbers(3, 4, 5, curry2(add)(60))
print(x)
```

```
(63, 64, 65)
```

Use case for currying #1

- Whenever another function requires a function that only takes one argument:

curry4.py

```
from operator import add
curry2 = lambda f: lambda x: lambda y: f(x, y)
make_adder = curry2(add)

def transform_numbers(num1, num2, num3, transform):
    return (transform(num1), transform(num2), transform(num3))

x = transform_numbers(3, 4, 5, lambda x: make_adder(60)(x))
print(x)
```

```
(63, 64, 65)
```

Use case for currying #2

- Turning a generalized function into a specialized function:

curry5.py

```
curry2 = lambda f: lambda x: lambda y: f(x, y)
def html_tag(tag_name, text):
    return "<" + tag_name + ">" + text + "</" + tag_name + ">"

p_tag = curry2(html_tag)("p")
x = p_tag("hello hello")
print(x)
```

```
<p>hello hello</p>
```

Use case for currying #2

curry6.py

```
def html_tag(tag_name, text):
    return "<" + tag_name + ">" + text + "</" + tag_name + ">"

import functools
p_tag = functools.partial(html_tag, "p")
x=p_tag("hello hello")

print(x)
```

```
<p>hello hello</p>
```

Why learn currying in Python?

🥦 It's good for you!

CPSC 46K introduces many concepts that aren't standard Python practice, but that show up in other languages.

Currying is a very common practice in functional programming languages like Haskell or Clojure.

Decorators

SECTION 5

A tracing function

- Let's make a higher-order tracing function.

decorate1.py

```python
def trace1(f):
    def traced(x):
        print("->", x)
        r = f(x)
        print("<-", r)
        return r
    return traced

def square(x):
    return x * x
square = trace1(square)
print(square(3))
```

```
-> 3
<- 9
9
```

decorate2.py

```python
def trace1(f):
    def traced(x):
        print("->", x)
        r = f(x)
        print("<-", r)
        return r
    return traced

@trace1
def square(x):
    return x * x

print(square(3))
```

```
-> 3
<- 9
9
```

A tracing decorator

- What if we always wanted a function to be traced?

```
@trace1
def square(x):
     return x * x
```

- That's equivalent to..

```
def square(x):
     return x * x
square = trace1(square)
```

General decorator syntax

The notation:

```
@ATTR
def aFunc(...):
    ...
```

- is essentially equivalent to:

```
def aFunc(...):
    ...
aFunc = ATTR(aFunc)
```

- `ATTR` can be any expression, not just a single function name.

Chaining Rule

SECTION 6 (TO BE ADDED)

Recursive Function

SECTION 7

Recursive Functions

- A function is recursive if the body of that function calls itself, either directly or indirectly.
- Recursive functions often operate on increasingly smaller instances of a problem.

Anatomy of a recursive function

- **Base case:** Evaluated without a recursive call (the smallest subproblem).
- **Recursive case:** Evaluated with a recursive call (breaking down the problem further.)
- **Conditional Statement:** to decide of it’s a base case.

Anatomy of a recursive function

sumofdigits.py

```
def sumOfDigits(n):
    if n==0: return 0                  # base case
    return n%10+sumOfDigits(n//10)  # recursive case

print(sumOfDigits(738))
```

```
18
```

Fibonacci Function

`F(n) = F(n-1) + F(n-2)`	**Recursive formula**
`F(0) = 1, F(1) = 1`	**Base condition**

Fibonacci Function

fibo.py

```
def fibo(n):
    if n==0 or n==1: return 1
    return fibo(n-1)+fibo(n-2)

print(fibo(10))
```

Iterative Version

fibo.py

```
def fibonacci(n):
    if n==0 or n==1: return 1
    fn = 0   # f(i)
    fn1 =1    # f(i-1)
    fn2 =1    # f(i-2)
    i = 2
    while i<=n:  # i>n
        fn = fn1 + fn2
        print("f(%d)=%d" % (i, fn))
        fn2 = fn1
        fn1 = fn
        i += 1

fibonacci(100)
```

The Recursive Leap of Faith

Is Fibonacci algorithm implemented correctly?

1. Verify the base case.
2. Treat fibo as a functional abstraction!
3. Assume that fibo(n-1) and fibo(n-2) are correct (← The leap)
4. Verify that fibo(n) is correct.

Write a recursive function

- Write the helper function first. Make sure you can pass enough arguments to the sub-cases and return the expected result.
- Write the main function after the helper function is complete.

binarySearch.py

```python
def bin(a, key):
    low = 0
    high = len(a)-1
    while low<=high:  # exit condition: cross over
        mid = (low+high)// 2
        if (a[mid]==key): return mid
        elif (a[mid]>key): high = mid -1
        else: low = mid +1
    return -1


x = [1, 5, 6, 8, 7, 12, 15]

print(bin(x, 3))
print(bin(x, 15))
print(bin(x, 1))
```

```
-1
6
0
```

binarySearch.py

```python
def binT(a, key):
    return binHelper(a, key, 0, len(a)-1)
    #return binHelper()

def binHelper(a, key, low, high):
    if (low > high): return -1
    mid = (low+high)//2
    if (a[mid]==key): return mid
    elif (a[mid]>key): return binHelper(a, key, low, mid-1)
    return binHelper(a, key, mid+1, high)

x = [1, 5, 6, 8, 7, 12, 15]

print(binT(x, 3))
print(binT(x, 15))
print(binT(x, 1))
```

```
-1
6
0
```

Mutual Recursion

SECTION 8

The Luhn Algorithm

Used to verify that a credit card number is valid

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit: if product of this doubling operation is greater than 9 (e.g., 7*2=14), then sum the digits of that product (e.g., 14: 1+5 = 5)
2. Take the sum of all the digits

Original	1	3	8	7	4	3
Processed						3

The Luhn Algorithm

Used to verify that a credit card number is valid

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit: if product of this doubling operation is greater than 9 (e.g., 7*2=14), then sum the digits of that product (e.g., 14: 1+5 = 5)
2. Take the sum of all the digits

Original	1	3	8	7	4	3
Processed					8	3

The Luhn Algorithm

Used to verify that a credit card number is valid

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit: if product of this doubling operation is greater than 9 (e.g., 7*2=14), then sum the digits of that product (e.g., 14: 1+5 = 5)
2. Take the sum of all the digits

Original	1	3	8	7	4	3
Processed				7	8	3

The Luhn Algorithm

Used to verify that a credit card number is valid

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit: if product of this doubling operation is greater than 9 (e.g., 7*2=14), then sum the digits of that product (e.g., 14: 1+5 = 5)
2. Take the sum of all the digits

Original	1	3	8	7	4	3
Processed			1+6=7	7	8	3

The Luhn Algorithm

Used to verify that a credit card number is valid

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit: if product of this doubling operation is greater than 9 (e.g., 7*2=14), then sum the digits of that product (e.g., 14: 1+5 = 5)
2. Take the sum of all the digits

Original	1	3	8	7	4	3
Processed		3	1+6=7	7	8	3

The Luhn Algorithm

Used to verify that a credit card number is valid

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit: if product of this doubling operation is greater than 9 (e.g., 7*2=14), then sum the digits of that product (e.g., 14: 1+5 = 5)
2. Take the sum of all the digits

Original	1	3	8	7	4	3
Processed	2	3	1+6=7	7	8	3

The Luhn Algorithm

Used to verify that a credit card number is valid

1. From the rightmost digit, which is the check digit, moving left, double the value of every second digit: if product of this doubling operation is greater than 9 (e.g., 7*2=14), then sum the digits of that product (e.g., 14: 1+5 = 5)
2. Take the sum of all the digits

Original	1	3	8	7	4	3		
Processed	2	3	1+6=7	7	8	3	=	30

The luhn sum of a valid credit card number is a multiple of 10

Calculating the Luhn Algorithm

sum_digits

luhn.py

```python
def sum_digits(n):
    if n<10:
        return n
    else:
        last = n % 10
        all_but_last = n // 10
        return last + sum_digits(all_but_last)
```

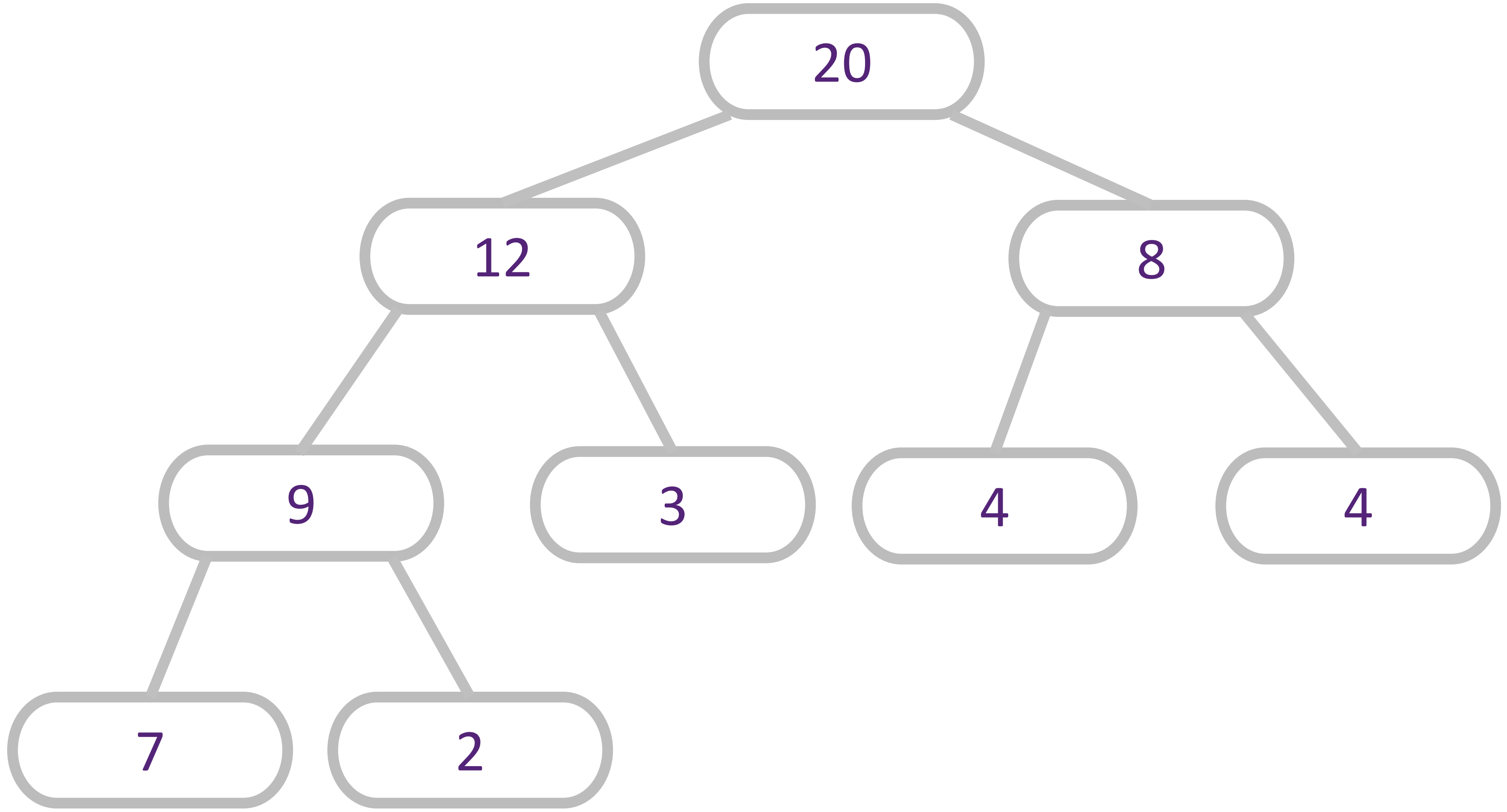
luhn.py

```
def luhn_sum(n):
    if n<10:
        return n
    else:
        last = n % 10
        all_but_last = n // 10
        return last + luhn_sum_double(all_but_last)

def luhn_sum_double(n):
    last = n % 10
    all_but_last = n // 10
    luhn_digit = sum_digits(last * 2)
    if n < 10:
        return luhn_digit
    else:
        return luhn_digit + luhn_sum(all_but_last)

sum = luhn_sum(138743)
print(sum)
```

Tree

SECTION 9

Trees



Trees

Recursive Description

- A tree has a **root label** and a list of **branches**
- Each **branch** is itself a tree
- A tree with zero branches is called a **leaf**
- A tree starts at the **root**

Relative Description

- Each location is a tree is called a **node**
- Each node has a **label** that can be any value
- One can be the **parent/child** of another
- The top node is the **root node**

Trees: Data Abstraction

- We want this constructor and selectors:

tree(label, branches)	Returns a tree with root label and list of branches
Label(tree)	Returns the root label of tree
branches(tree)	Returns the branches of tree (each a tree)
is_leaf(tree)	Returns true if tree is a leaf node

tree.py

```python
def tree(label, branches=[]):
    return [label]+ list(branches)
def label(tree):
    return tree[0]
def branches(tree):
    return tree[1:]
def is_leaf(tree):
    return len(branches(tree)) == 0
t = tree(3, [
            tree(1),
            tree(2, [
                tree(1),
                tree(1)
            ])
        ])

lx = label(t); print(lx)  # 3
tx = is_leaf(branches(t)[0]); print(tx) # true
```

```
3
true
```

Tree Processing

- A tree is a recursive structure.
- Each tree has:
  - A Label
  - 0 or more branches, each a tree
- Recursive Structure means recursive algorithms.

Tree Processing: Counting Leaves

Tree_count_leaves.py

```
def count_leaves(t):
    if is_leaf(t):
        return 1
    else:
        leaves_under = 0
        for b in branches(t):
            leaves_under += count_leaves(b)
        return leaves_under
```

Tree Processing: Counting Leaves (Sum)

Tree_count_leaves.py

```
def count_leaves_sum(t):
    if is_leaf(t):
        return 1
    else:
        leaves_unders = [count_leaves_sum(b) for b in branches(t)]
        return sum(leaves_unders)
```

Tree Duplication

- Copy from one tree to the other.
- It is a recursive function.

Copy

tree_copy.py

```
def copy(t):
    if is_leaf(t):
        return tree(label(t))
    else:
        return tree(label(t), [tree(b) for b in branches(t)])
```

```
[3, [[1]], [[2, [1], [1]]]]
```

Print

- Print a tree is also a recursive function

tree_copy.py

```
def print_tree(t, indent=0):
    print(indent*" ", label(t))
    for b in branches(t):
        print_tree(b, indent+3)
```

```
3
   1
   2
      1
      1
```

Sum of Lists

tree_leaves.py

```
def sum_list(*args):
    s = []
    for alist in args:
        s += alist
    return s

print(sum_list([1], [2, 3], [4]))
```

```
[1, 2, 3, 4]
```

tree_leaves.py

```python
def leaves(t):
    if is_leaf(t): return t
    else:
        leaf_labels = [leaves(b) for b in branches(t)]
        return sum(leaf_labels, [])

t = tree(20, [tree(12, [tree(9, [tree(7), tree(2)]), tree(8, [tree(4), tree(4)])])])
print_tree(t)
print(leaves(t))
```

```
20
  12
    9
      7
      2
    8
      4
      4
[7, 2, 4, 4]
```

Count Paths

- Return the number of paths from the root to any node in `t` for which the labels along the path sum to total.

tree_leaves.py

```
def count_paths(t, total):
    if label(t) == total:
        found = 1
    else:
        found = 0
    return found + sum([count_paths(b, total-label(t)) for b in branches(t)])
tx = tree(3, [tree(-1), tree(1, [tree(2, [tree(1)]), tree(3)]), tree(1, [tree(1)])])

print_tree(tx)
print(count_paths(tx, 3))
print(count_paths(tx, 4))
print(count_paths(tx, 5))
print(count_paths(tx, 6))
print(count_paths(tx, 7))
```

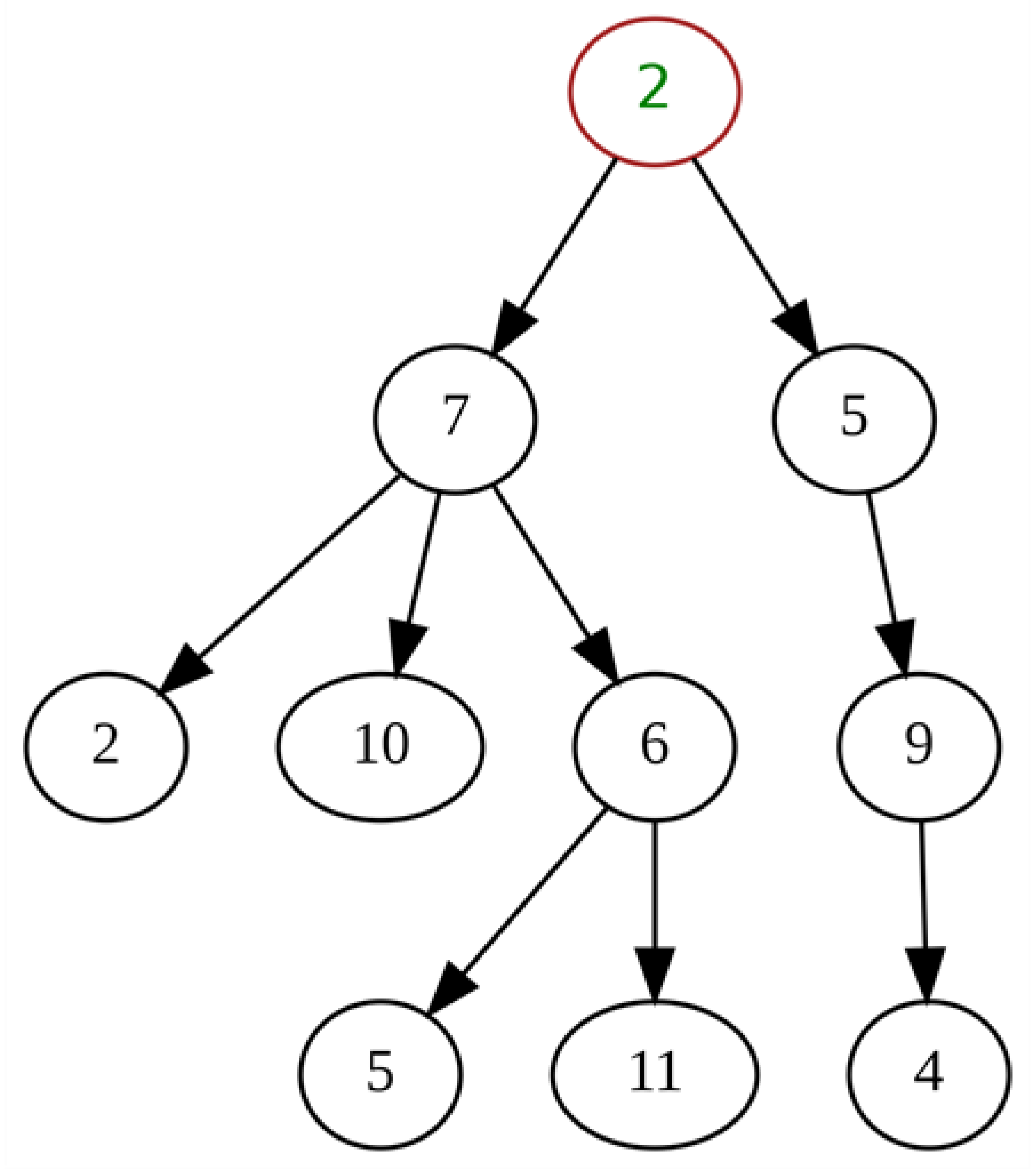
```
3
  -1
  1
    2
      1
    3
  1
    1
1
2
1
1
2
```

Three Layers of Abstraction for Tree

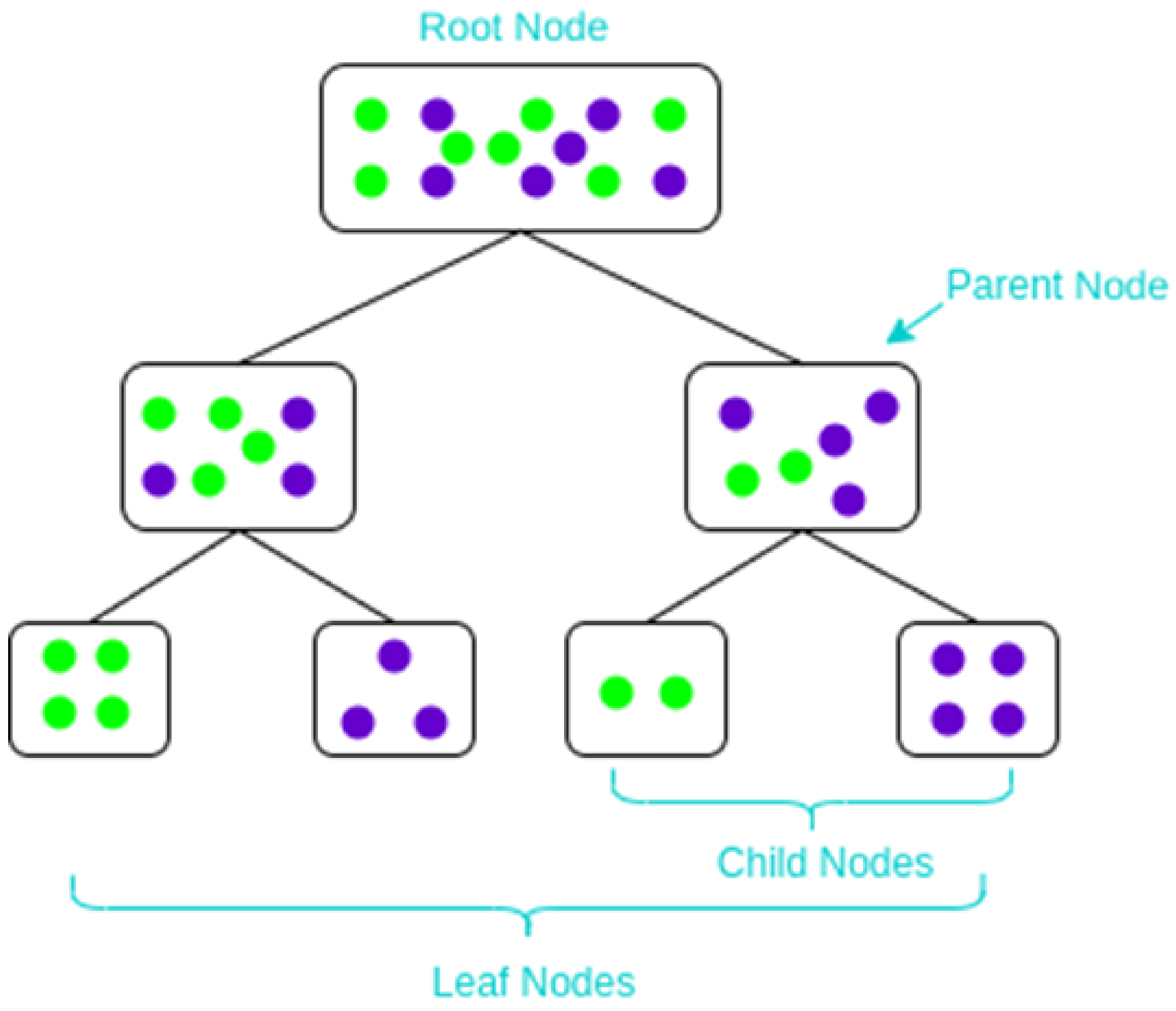
- Data Storage and Data Structure:
  - `1, 2, 3, “a”, “b”, “c”  (int, str)`
  - `[1, 2, 3] (list)`
- Abstract Data Type: `tree(), branches(),`
  `label(), is_leaf()`
- Data Operations (Functions): `copy(t), print_tree(t),`
  `leaves(t), count_paths(t)`

Type of Trees

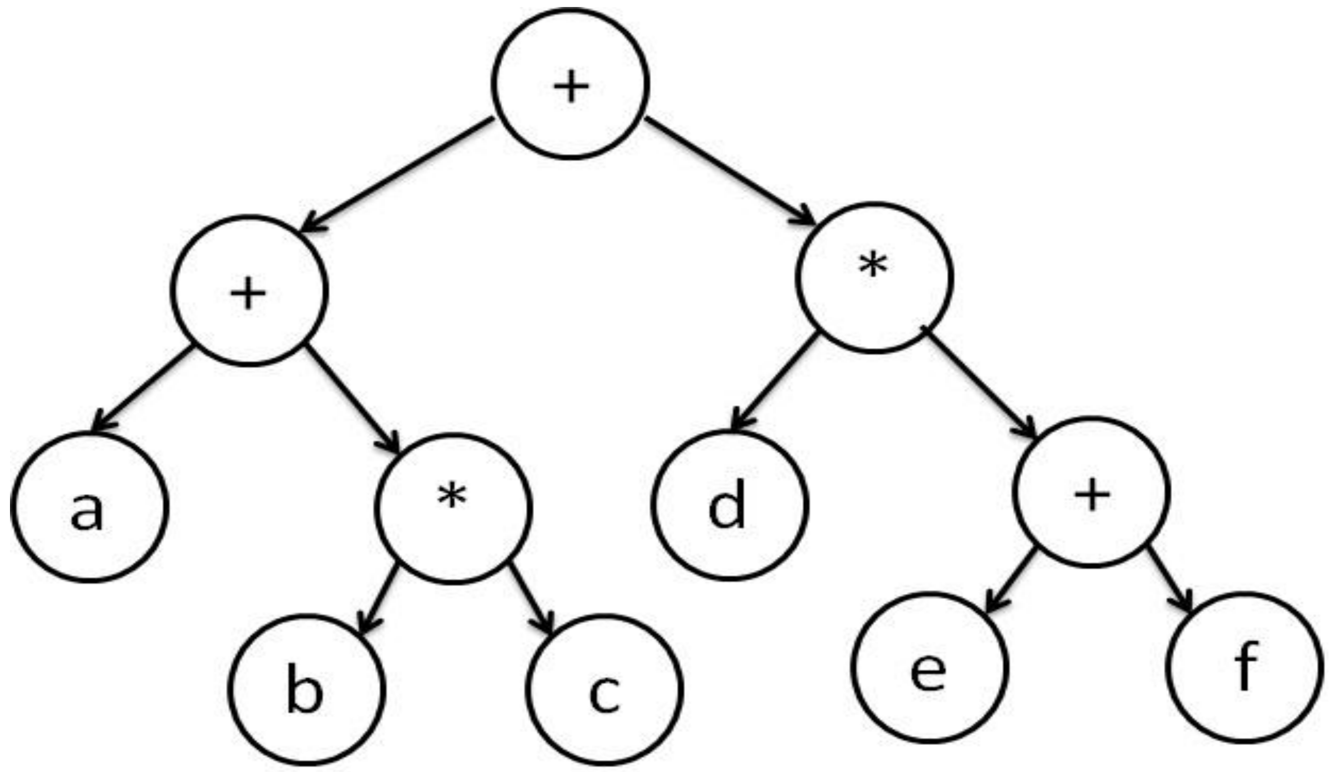
SECTION 10



Data Tree

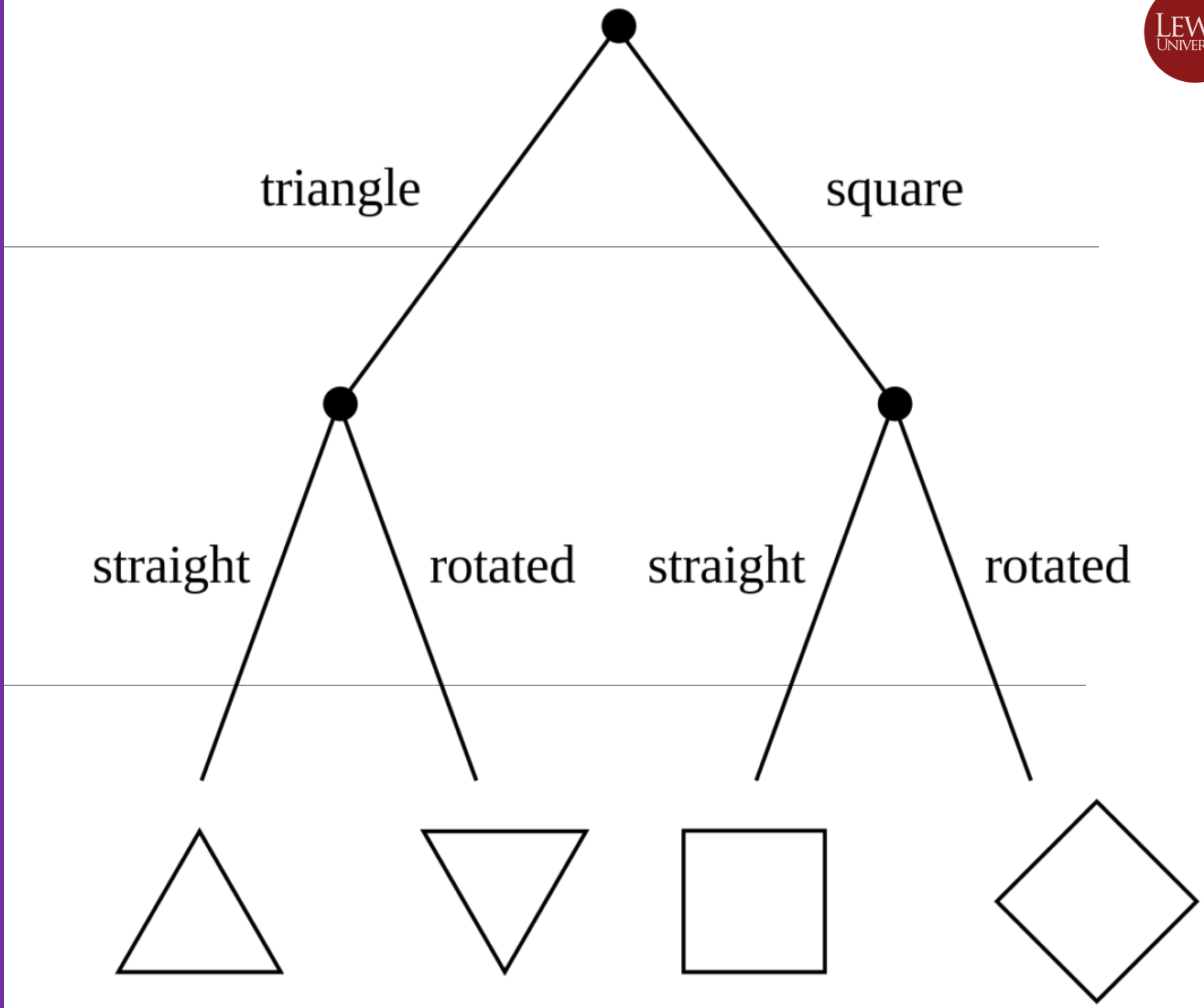


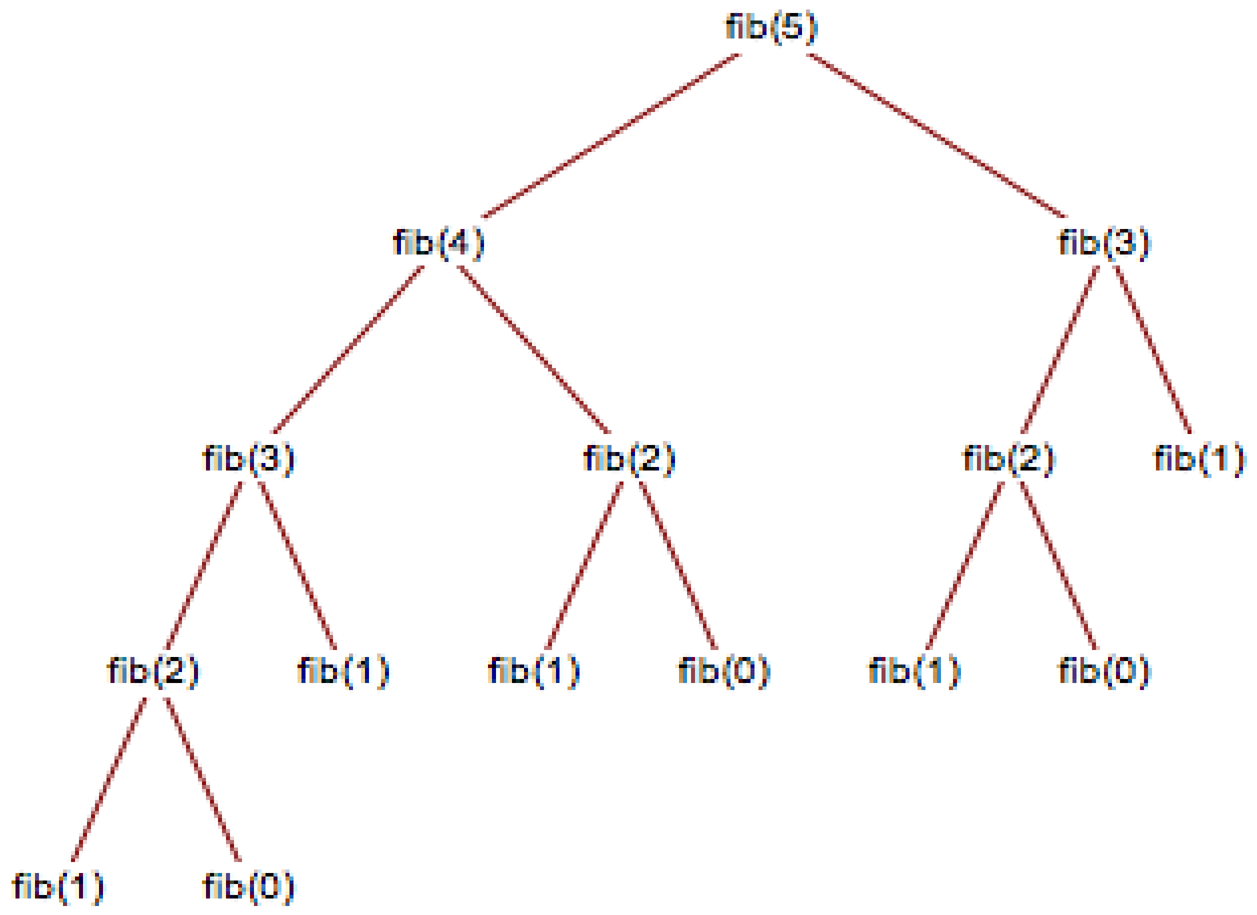
Component Tree



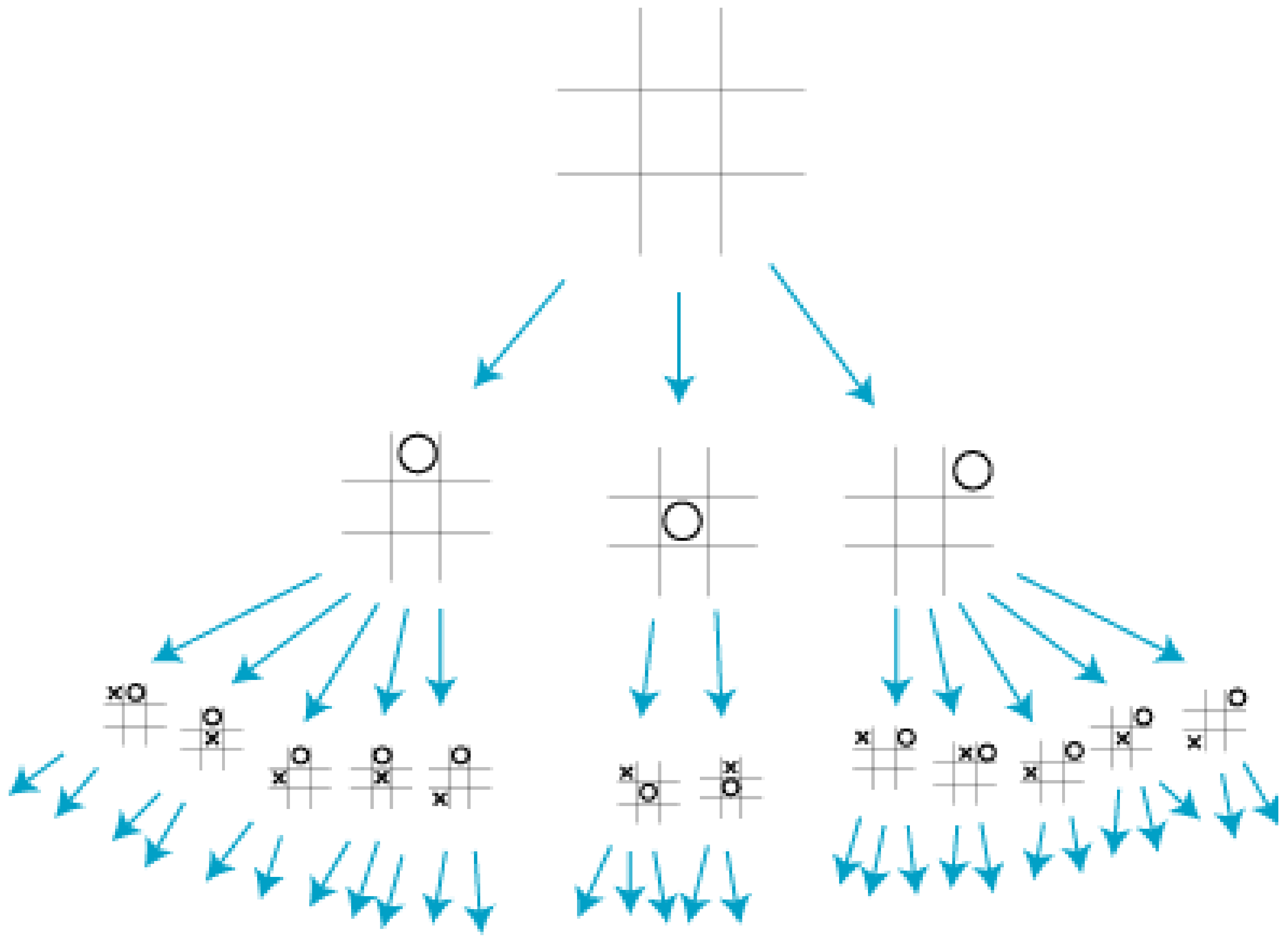
Expression Tree

Decision Tree



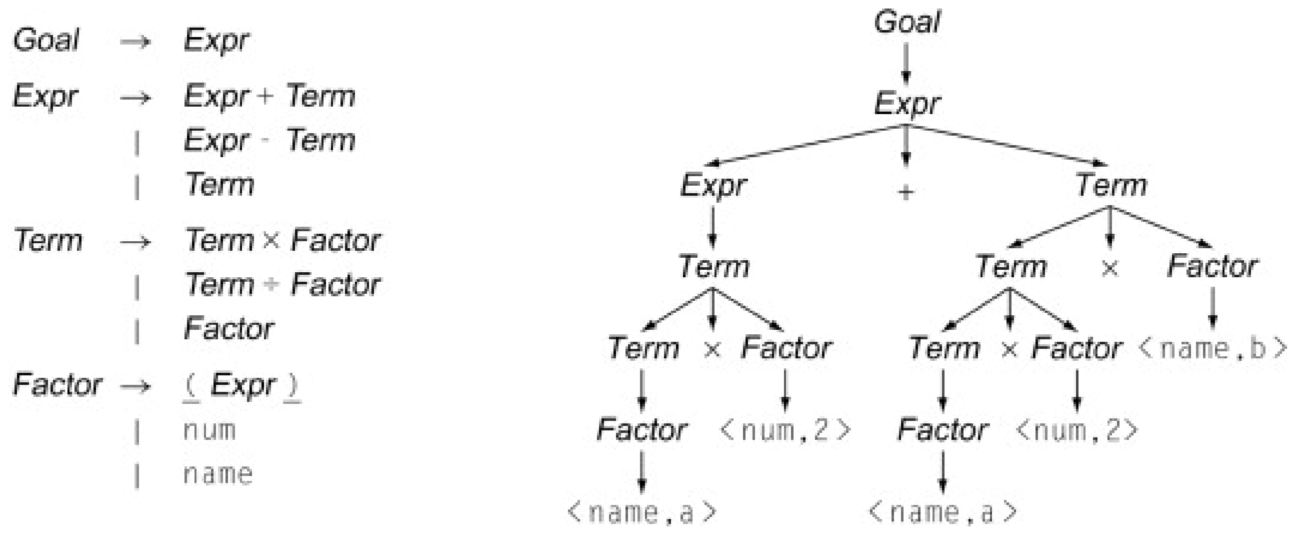


Recursion Tree



Game Evaluation Tree

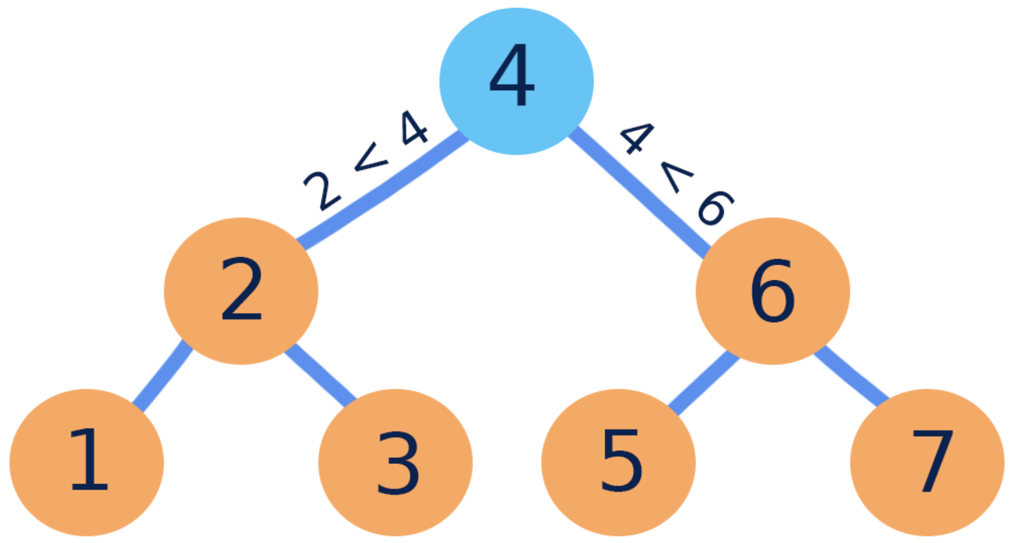
Parsing Tree



```
Goal   → Expr
Expr   → Expr + Term
       | Expr - Term
       | Term
Term   → Term × Factor
       | Term ÷ Factor
       | Factor
Factor → ( Expr )
       | num
       | name
```

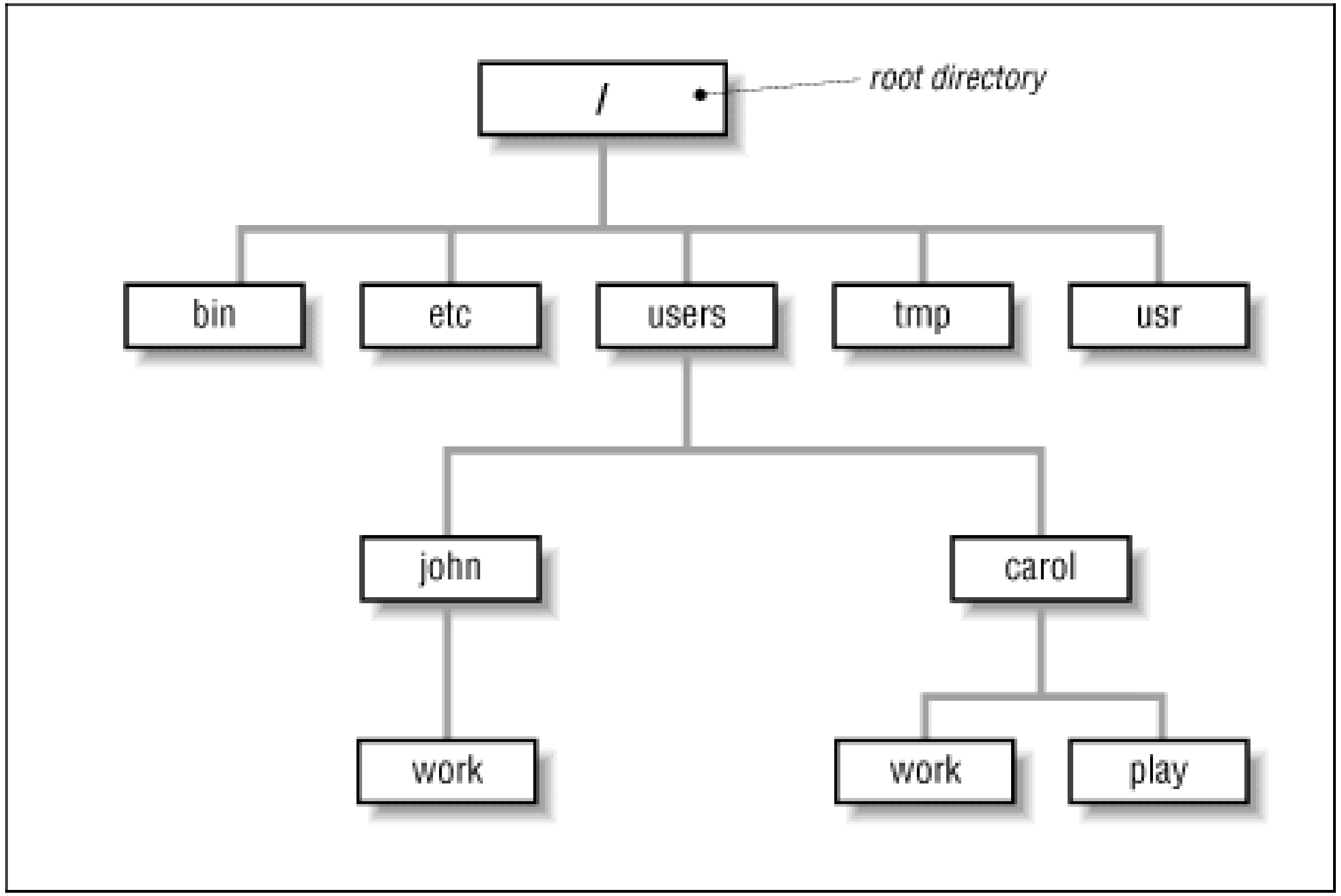
(a) Classic Expression Grammar

(b) Parse Tree for a × 2 + a × 2 × b



In Order Traversal: 1 2 3 4 5 6 7

Binary Search Tree



Directory System Tree

Introduction to Pushdown Automata

SECTION 11

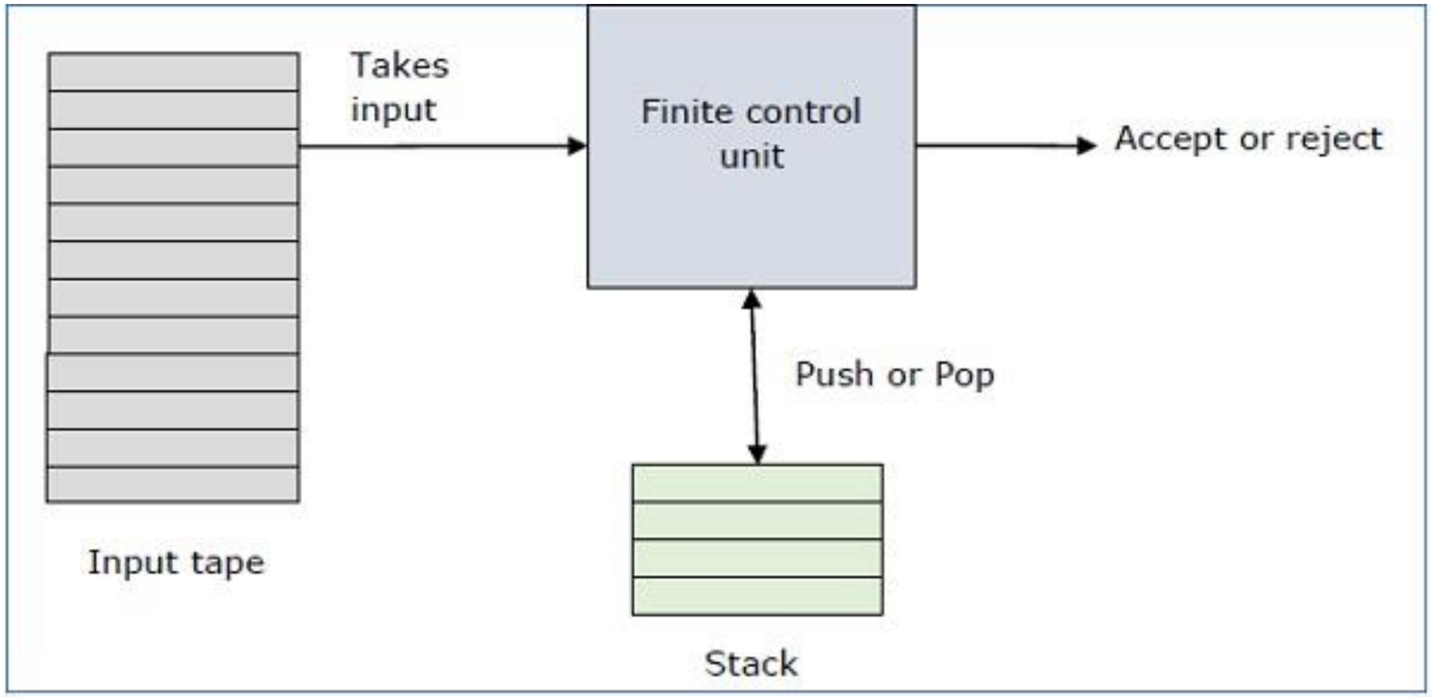
Basic Structure of PDA

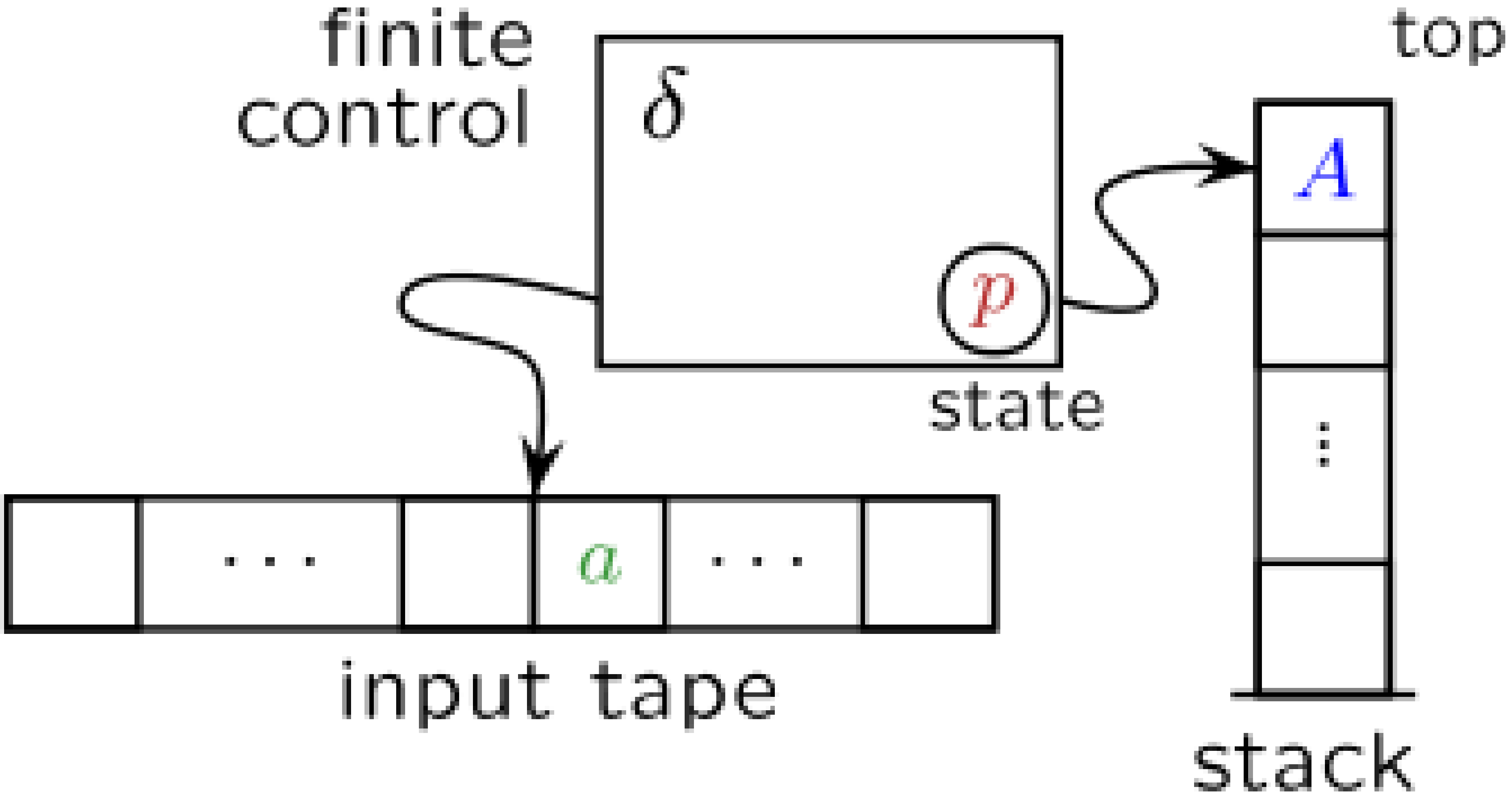
- A pushdown automaton is a way to implement a context-free grammar in a similar way we design DFA for a regular grammar. A DFA can remember a finite amount of information, but a PDA can remember an infinite amount of information.
- Basically a pushdown automaton is –

"Finite state machine" + "a stack"

PDA

A PDA may or may not read an input symbol, but it has to read the top of the stack in every transition.




finite control
δ
top
A
p
state
$\vdots$
$\cdots$
a
$\cdots$
input tape
stack

Pushdown Automata

- A pushdown automaton has three components –
  - an input tape,
  - a control unit, and
  - a stack with infinite size.
- The stack head scans the top symbol of the stack.
- A stack does two operations –
  - **Push** – a new symbol is added at the top.
  - **Pop** – the top symbol is read and removed.

Formal Description of PDA

A PDA can be formally described as a 7-tuple (Q, ∑, S, δ, q_0, I, F) –

- **Q** is the finite number of states
- **∑** is input alphabet
- **S** is stack symbols
- **δ** is the transition function: Q × (∑ ∪ {ε}) × S × Q × S*
- **q_0** is the initial state (q_0 ∈ Q)
- **I** is the initial stack top symbol (I ∈ S)
- **F** is a set of accepting states (F ∈ Q)

Stack

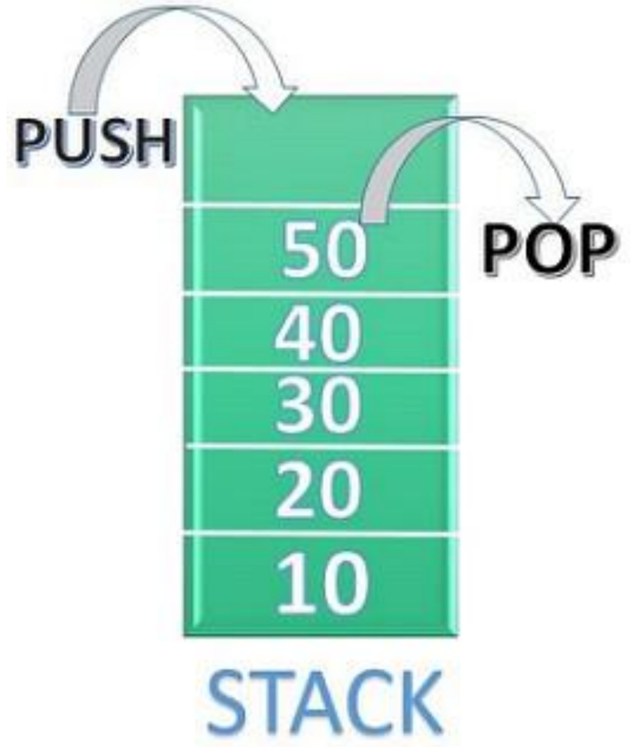
SECTION 12

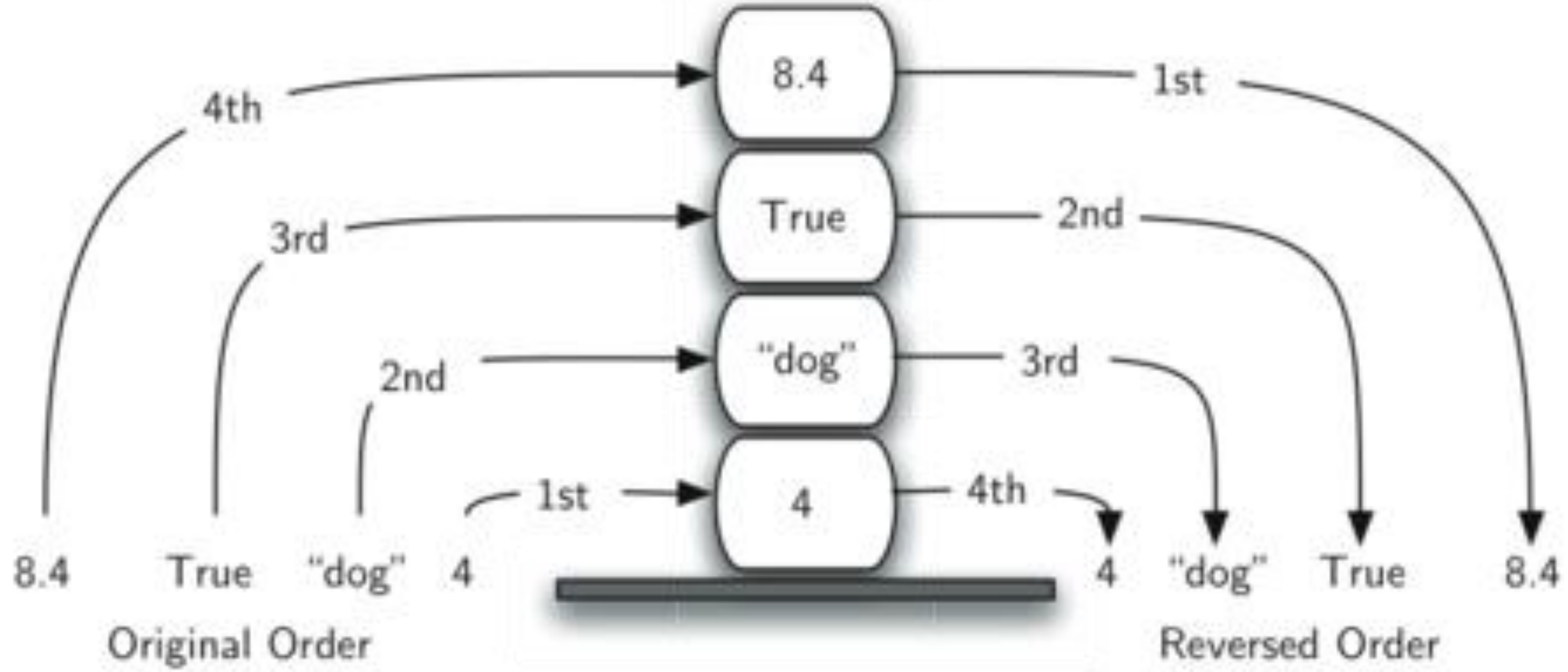
What is a Stack?

- A **stack** (sometimes called a “push-down stack”) is an ordered collection of items where the addition of new items and the removal of existing items always takes place at the same end.
- This end is commonly referred to as the “top.” The end opposite the top is known as the “base.”

Stack

A stack is a linear Structure in which items may be added or removed only at one end, called the top of the stack. This means, in particular, the elements are removed from a stack in the reverse order of that which they are inserted in to the stack. The stacks are also called "Last-in First -Out (LIFO) " list.




4th
3rd
2nd
1st
8.4
True
"dog"
4
1st
2nd
3rd
4th
8.4
True
"dog"
4
4
"dog"
True
8.4
Original Order
Reversed Order

Primary operations defined on a stack

- **Push:** Insert an element at the top of the list.
- **Pop :** Delete an element from the top of the list.
- **Peek :** Return value of topmost element.

The Stack Abstract Data Type

The stack operations are given below.

- 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 and returns the item. The stack is modified.
- peek() returns the top item from the stack but does not remove it. It needs no parameters. The stack is not modified.

The Stack Abstract Data Type

- isEmpty() 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.isEmpty()	[]	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.isEmpty()	[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

Python Stack Using List

SECTION 13

Implementation of Stack

The stack data structure can be implemented in two different ways

1. Using array [Static Implementation]

2. Using Linked List [Dynamic Implementation]

Using Primitive Collections

- Python, as in any object-oriented programming language, the implementation of choice for an abstract data type such as a stack is the creation of a new class.
- The stack operations are implemented as methods. Further, to implement a stack, which is a collection of elements, it makes sense to utilize the power and simplicity of the **primitive collections** provided by Python. We will use a list.

Using Primitive Collections

- Recall that the list class in Python provides an ordered collection mechanism and a set of methods. For example, if we have the **list** [2,5,3,6,7,4], we need only to decide which end of the list will be considered the top of the stack and which will be the base.
- Once that decision is made, the operations can be implemented using the list methods such as **append** and **pop**.

Stack Class in Python

Demo Program: stack.py

- The following stack implementation (stack.py) assumes that the end of the list will hold the top element of the stack.
- As the stack grows (as push operations occur), new items will be added on the end of the list. pop operations will manipulate that same end.

Stack.py

```python
class Stack:
    def __init__(self):
        self.items = []
    def isEmpty(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[len(self.items)-1]
    def size(self):
        return len(self.items)

s=Stack()

print(s.isEmpty())
s.push(4)
s.push('dog')
print(s.peek())
s.push(True)
print(s.size())
print(s.isEmpty())
s.push(8.4)
print(s.pop())
print(s.pop())
print(s.size())
```

```
True
dog
3
False
8.4
True
2
```

Stack Class in Python

Demo Program: Stack2.py

Stack2.py

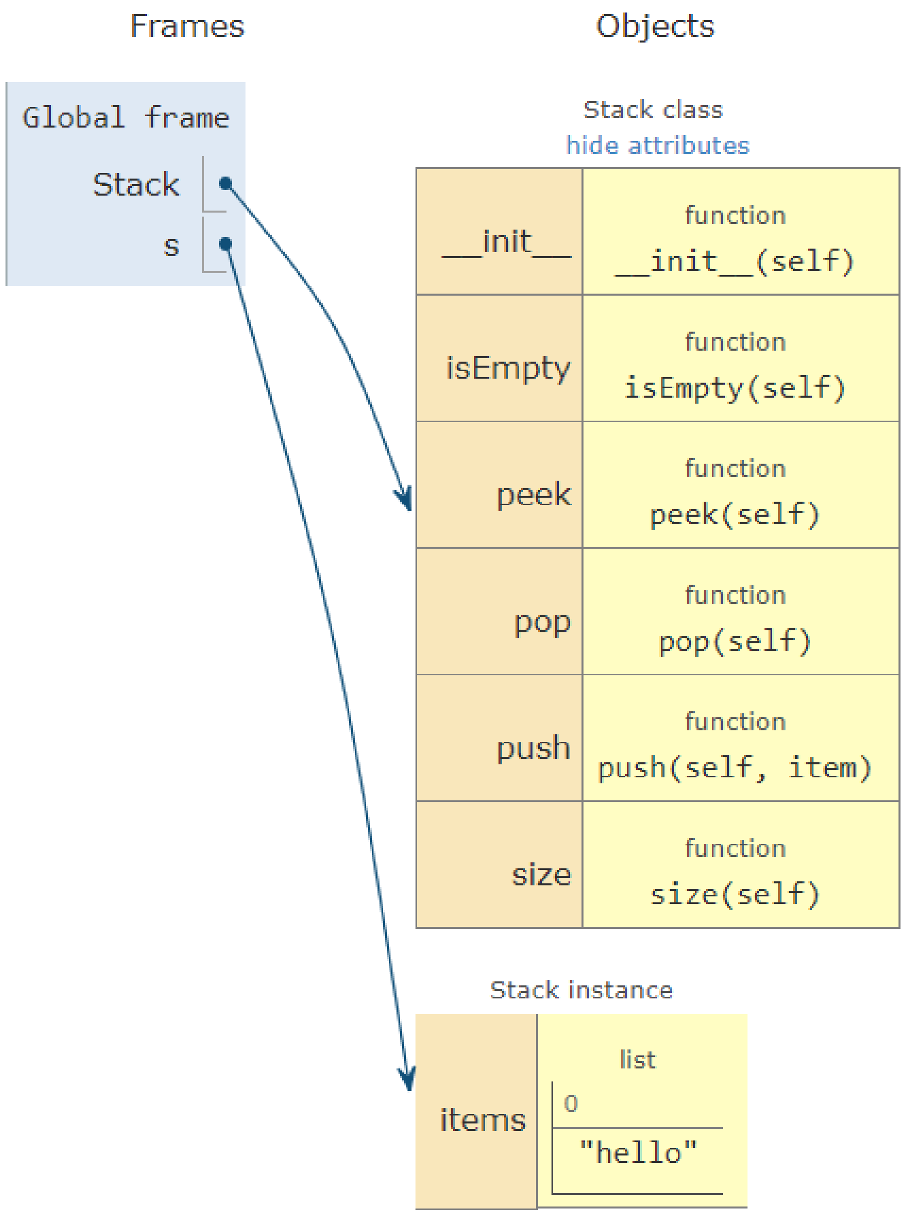
```python
class Stack:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def push(self, item):
        self.items.insert(0,item)
    def pop(self):
        return self.items.pop(0)
    def peek(self):
        return self.items[0]
    def size(self):
        return len(self.items)

s = Stack()
s.push('hello')
s.push('true')
print(s.pop())
```

```
true
```

Stack Class in Python

Demo Program: Stack2.py

- This ability to change the physical implementation of an abstract data type while maintaining the logical characteristics is an example of abstraction at work. However, even though the stack will work either way, if we consider the performance of the two implementations, there is definitely a difference. Recall that the append and **pop()** operations were both **O(1)**.
- This means that the first implementation will perform push and pop in constant time no matter how many items are on the stack. The performance of the second implementation suffers in that the **insert(0)** and **pop(0)** operations will both require **O(n)** for a stack of size **n**.
- Clearly, even though the implementations are logically equivalent, they would have very different timings when performing benchmark testing.



http://interactivepython.org/runestone/static/pythonds/BasicDS/ImplementingaStackinPython.html

Parentheses

SECTION 14

Simple Balanced Parentheses

- We now turn our attention to using stacks to solve real computer science problems. You have no doubt written arithmetic expressions such as

```
(5+6)∗(7+8)/(4+3)(5+6)∗(7+8)/(4+3)
```

- where parentheses are used to order the performance of operations. You may also have some experience programming in a language such as Lisp with constructs like

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

- This defines a function called **square** that will return the square of its argument n. Lisp is notorious for using lots and lots of parentheses.

Balanced parentheses

- In both of these examples, parentheses must appear in a balanced fashion. Balanced parentheses means that each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested. Consider the following correctly balanced strings of parentheses:

```
(( )( )( )( ))
(((( ))))
(( )((( ))( )))
```

Balanced parentheses

- Compare those with the following, which are not balanced:

```
((((((( ))
( )))
(( )( )(( )
```

- The ability to differentiate between parentheses that are correctly balanced and those that are unbalanced is an important part of recognizing many programming language structures.

Balanced parentheses

- The challenge then is to write an algorithm that will read a string of parentheses from left to right and decide whether the symbols are balanced. To solve this problem we need to make an important observation. As you process symbols from left to right, the most recent opening parenthesis must match the next closing symbol (see Figure 4).
- Also, the first opening symbol processed may have to wait until the very last symbol for its match. Closing symbols match opening symbols in the reverse order of their appearance; they match from the inside out. This is a clue that stacks can be used to solve the problem.

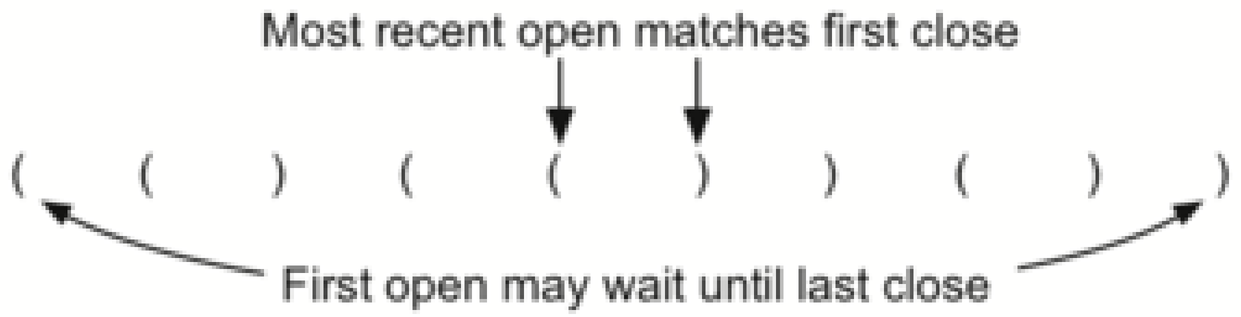

Most recent open matches first close
(( ) (( )) ( ))
First open may wait until last close

Figure 4

Balanced parentheses

- Once you agree that a stack is the appropriate data structure for keeping the parentheses, the statement of the algorithm is straightforward.
- Starting with an empty stack, process the parenthesis strings from left to right. If a symbol is an opening parenthesis, push it on the stack as a signal that a corresponding closing symbol needs to appear later. If, on the other hand, a symbol is a closing parenthesis, pop the stack.
- As long as it is possible to pop the stack to match every closing symbol, the parentheses remain balanced. If at any time there is no opening symbol on the stack to match a closing symbol, the string is not balanced properly.
- At the end of the string, when all symbols have been processed, the stack should be empty. The Python code to implement this algorithm is shown in **parcheckers.py**.

Demo Program: parcheckers.py

- This function, **parChecker**, assumes that a **Stack** class is available and returns a boolean result as to whether the string of parentheses is balanced. Note that the boolean variable balanced is initialized to **True** as there is no reason to assume otherwise at the start. If the current symbol is (, then it is pushed on the stack (lines 9–10).
- Note also in line 15 that pop simply removes a symbol from the stack. The returned value is not used since we know it must be an opening symbol seen earlier.
- At the end (lines 19–22), as long as the expression is balanced and the stack has been completely cleaned off, the string represents a correctly balanced sequence of parentheses.

```python
from Stack import Stack

def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol == "(":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                s.pop()
        index = index + 1
    if balanced and s.isEmpty():
        return True
    else:
        return False

print(parChecker('((()))'))
print(parChecker('(()'))
```

# parcheckers.py (First half)

```
True
False
```

Balanced Symbols (A General Case)

SECTION 15

Balanced Symbols (A General Case)

- The balanced parentheses problem shown above is a specific case of a more general situation that arises in many programming languages. The general problem of balancing and nesting different kinds of opening and closing symbols occurs frequently. For example, in Python square brackets, [and], are used for lists; curly braces, { and }, are used for dictionaries; and parentheses, (and), are used for tuples and arithmetic expressions. It is possible to mix symbols as long as each maintains its own open and close relationship. Strings of symbols such as

```
{ { ( [ ] [ ] ) } ( ) }
[ [ { { ( ( ) ) } } ] ]
[ ] [ ] [ ] ( ) { }
```

- are properly balanced in that not only does each opening symbol have a corresponding closing symbol, but the types of symbols match as well.

Balanced Symbols (A General Case)

- Compare those with the following strings that are not balanced:

```
( [ ) ]
( ( ( ) ] ) )
[ { ( ) ]
```

- The simple parentheses checker from the previous section can easily be extended to handle these new types of symbols. Recall that each opening symbol is simply pushed on the stack to wait for the matching closing symbol to appear later in the sequence. When a closing symbol does appear, the only difference is that we must check to be sure that it correctly matches the type of the opening symbol on top of the stack. If the two symbols do not match, the string is not balanced. Once again, if the entire string is processed and nothing is left on the stack, the string is correctly balanced.

Demo Program: parcheckers2.py

- The Python program to implement this is shown in **parcheckers2.py**.
- The only change appears in line 16 where we call a helper function, matches, to assist with symbol-matching.
- Each symbol that is removed from the stack must be checked to see that it matches the current closing symbol.
- If a mismatch occurs, the **boolean** variable balanced is set to False.

Demo Program: parcheckers2.py

- These two examples show that stacks are very important data structures for the processing of language constructs in computer science.
- Almost any notation you can think of has some type of nested symbol that must be matched in a balanced order.
- There are a number of other important uses for stacks in computer science. We will continue to explore them in the next sections.

```python
from stack import Stack                                                    # parcheckers2.py
def parChecker(symbolString):
    s = Stack()
    balanced = True
    index = 0
    while index < len(symbolString) and balanced:
        symbol = symbolString[index]
        if symbol in "([{":
            s.push(symbol)
        else:
            if s.isEmpty():
                balanced = False
            else:
                top = s.pop()
                if not matches(top, symbol):
                    balanced = False
        index = index + 1
    if balanced and s.isEmpty():
        return True
    else:
        return False

def matches(open, close):
    opens = "([{"
    closers = ")]}"
    return opens.index(open) == closers.index(close)

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

```
True
False
```

Algebraic Expressions

SECTION 16

Algebraic Expressions

- An algebraic expression can be defined as follows...

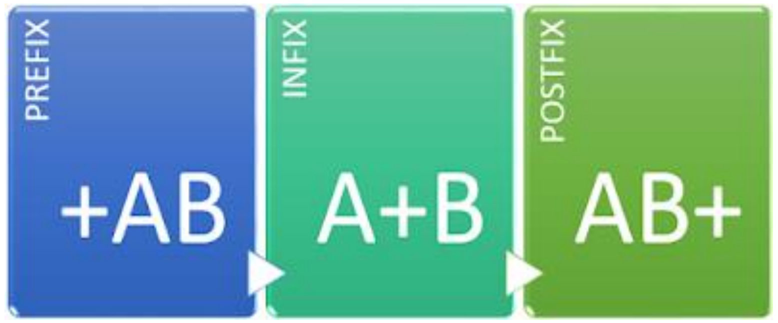
  An algebraic expression is a combination of operands and operators that represents a specific value.

- Example : A = B + C; denote an expression in which there are 3 operands A, B, C and two operator + and =
- An **operator** is a symbol that tells the computer to perform certain mathematical or logical manipulations. Operators are used in program to manipulate data and variables
- **Operands** are the values on which the operators can perform the task.

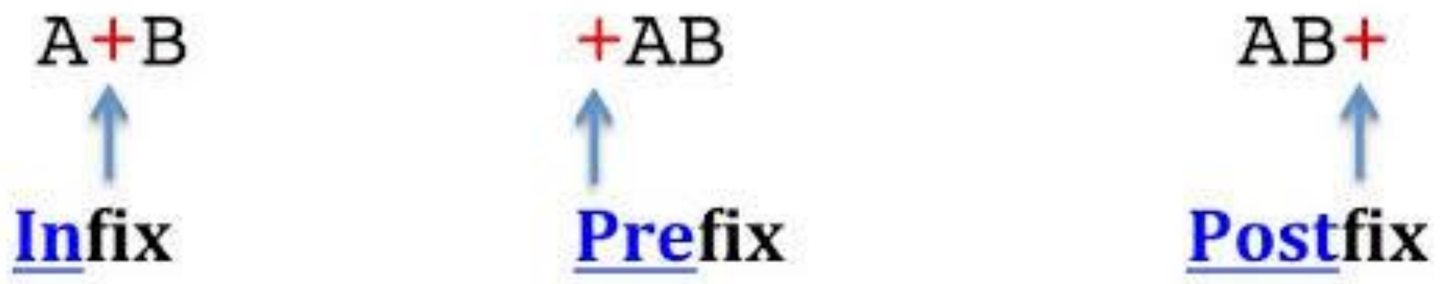
Types of Algebraic Expression

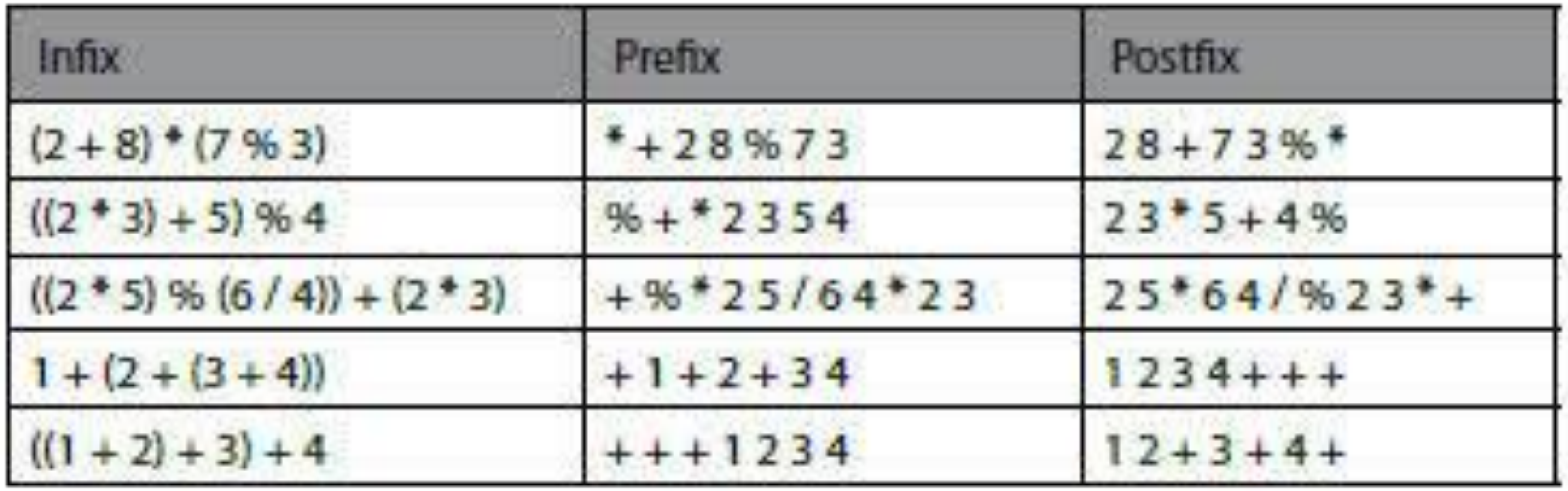
An algebraic expression can be represented in three different ways based on the operator position. They are as follows...

1. **Prefix Expression**
2. **Infix Expression**
3. **Postfix Expression**



Infix, Prefix and Postfix Expressions





Infix	Prefix	Postfix
(2 + 8) * (7 % 3)	* + 2 8 % 7 3	2 8 + 7 3 % *
((2 * 3) + 5) % 4	% + * 2 3 5 4	2 3 * 5 + 4 %
((2 * 5) % (6 / 4)) + (2 * 3)	+ % * 2 5 / 6 4 * 2 3	2 5 * 6 4 / % 2 3 * +
1 + (2 + (3 + 4))	+ 1 + 2 + 3 4	1 2 3 4 + + +
((1 + 2) + 3) + 4	+ + + 1 2 3 4	1 2 + 3 + 4 +

Precedence Levels

SECTION 17

Precedence Level

- Each operator has a precedence level.
- Operators of higher precedence are used before operators of lower precedence.
- The only thing that can change that order is the presence of parentheses. The precedence order for arithmetic operators places multiplication and division above addition and subtraction. If two operators of equal precedence appear, then a left-to-right ordering or associativity is used.
- Let’s interpret the troublesome expression A + B * C using operator precedence.
- B and C are multiplied first, and A is then added to that result. (A + B) * C would force the addition of A and B to be done first before the multiplication.
- In expression A + B + C, by precedence (via associativity), the leftmost + would be done first.

Fully Parenthesized Expressions

- One way to write an expression that guarantees there will be no confusion with respect to the order of operations is to create what is called a fully parenthesized expression.
- This type of expression uses one pair of parentheses for each operator. The parentheses dictate the order of operations; there is no ambiguity. There is also no need to remember any precedence rules.
- The expression A + B * C + D can be rewritten as ((A + (B * C)) + D) to show that the multiplication happens first, followed by the leftmost addition. A + B + C + D can be written as (((A + B) + C) + D) since the addition operations associate from left to right.

Postfix Conversion

SECTION 18

infix to postfix conversion using Stack

Step 1: Scan all the symbols one by one from left to right in the given Infix Expression and repeat steps 2-5

Step 2: If the scanned symbol is an **operand**, then place directly in the postfix expression

Step 3: If the scanned symbol is **left parenthesis '('**, push it onto the **STACK**.

Step 4: If the scanned symbol is an **operator (✕)**, then:

> Repeatedly pop from **STACK** and add to postfix expression each operator(On the top of STACK) , which has the same precedence as or higher precedence than (✕)
>
> Add (✕) to Stack

Step 5: If the symbol scanned is a **right parenthesis ')'**

> Repeatedly pop from STACK and add to postfix expression each operator(On the top of STACK) , till we get the matching left parenthesis.
>
> Remove the left parenthesis [Do not add the left parenthesis to postfix expression]

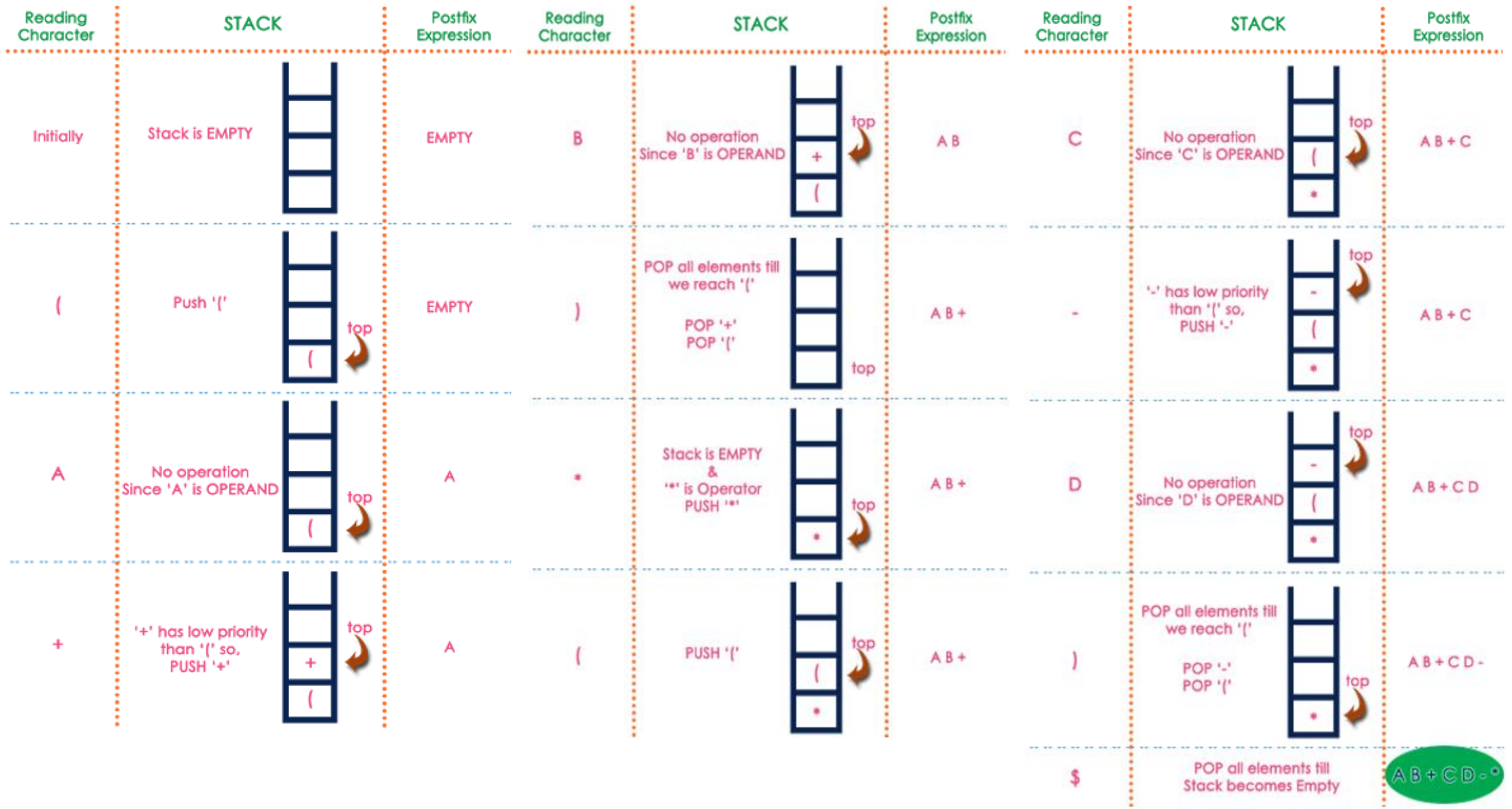
Step 6: EXIT

Example

Consider the following Infix Expression...

(A + B) * (C - D)

- The given infix expression can be converted into postfix expression using Stack data Structure as follows...



Reading Character	STACK	Postfix Expression
Initially	Stack is EMPTY	EMPTY
(	Push '(' (top	EMPTY
A	No operation Since 'A' is OPERAND (top	A
+	'+' has low priority than '(' so, PUSH '+' (+ top	A

Reading Character	STACK	Postfix Expression
B	No operation Since 'B' is OPERAND (+ top	A B
)	POP all elements till we reach '(' POP '+' POP '(' top	A B +
*	Stack is EMPTY & '*' is Operator PUSH '*' * top	A B +
(	PUSH '(' * (top	A B +

Reading Character	STACK	Postfix Expression
C	No operation Since 'C' is OPERAND * (top	A B + C
-	'-' has low priority than '(' so, PUSH '-' * (- top	A B + C
D	No operation Since 'D' is OPERAND * (- top	A B + C D
)	POP all elements till we reach '(' POP '-' POP '(' * top	A B + C D -
$	POP all elements till Stack becomes Empty	A B + C D - *

Python program to convert infix to postfix

Demo Program: infix_to_postfix.py

infix_to_postfix.py

```
class Stack:
    def __init__(self):
        self.items = []
    def isEmpty(self):
        return self.items == []
    def push(self, item):
        self.items.append(item)
    def pop(self):
        if self.isEmpty():
            return None
        return self.items.pop()
    def peek(self):
        if self.isEmpty():
            return None
        return self.items[len(self.items)-1]
    def size(self):
        return len(self.items)
    def __str__(self):
        return str(self.items)
```

Precedence of Algebraic Operators

infix_to_postfix.py

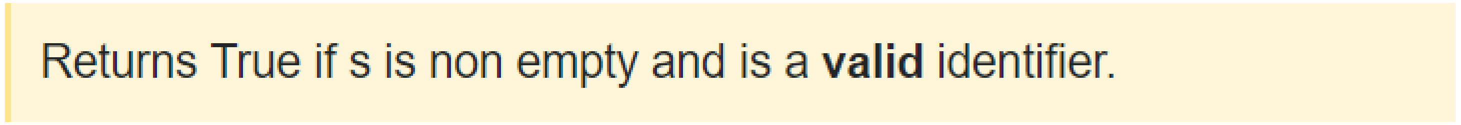
```
precedence = {}
precedence['*'] = 3
precedence['/'] = 3
precedence['+'] = 2
precedence['-'] = 2
precedence['('] = 1
```

infix_to_postfix.py

```python
def convert(expression):        # convert function from infix string to postfix list
    return __convert(expression.split())
def __convert(tokens):
    postfix = []
    opstack = Stack()
    for token in tokens:
        if token.isidentifier():
            postfix.append(token)
        elif token == '(':
            opstack.push(token)
        elif token == ')':
            while True:
                temp = opstack.pop()
                if temp is None or temp == '(':
                    break
                elif not temp.isidentifier():
                    postfix.append(temp)
        else:   # must be operator
            if not opstack.isEmpty():
                temp = opstack.peek()

                while not opstack.isEmpty() and precedence[temp] >= precedence[token] and token.isidentifier():
                    postfix.append(opstack.pop())
                    temp = opstack.peek()
            opstack.push(token)
    while not opstack.isEmpty():
        postfix.append(opstack.pop())
    return postfix
```

Python string function isidentifier()

- Python’s string function to check is a string is in valid ID format.



Returns True if s is non empty and is a **valid** identifier.

- Important function when we are processing tokens.

Rules for writing identifiers

1. Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_). Names like myClass, var_1 and print_this_to_screen, all are valid example.
2. An identifier cannot start with a digit. 1variable is invalid, but variable1 is perfectly fine.
3. Keywords cannot be used as identifiers.

```
>>> global = 1
  File "<interactive input>", line 1
    global = 1
           ^
SyntaxError: invalid syntax
```

4. We cannot use special symbols like !, @, #, $, % etc. in our identifier.

```
>>> a@ = 0
  File "<interactive input>", line 1
    a@ = 0
     ^
SyntaxError: invalid syntax
```

5. Identifier can be of any length.

toString()

infix_to_postfix.py

```
def toString(items):
    string = ""
    for i in range(len(items)):
        string = string + str(items[i])
    return string
```

Examples

infix_to_postfix.py

```
list1 = convert("A + B")
print("Postfix: ", toString(list1))
list2 = convert("A + B * C")
print("Postfix: ", toString(list2))
list3 = convert("A * ( B + C ) + D")
print("Postfix: ", toString(list3))
list4 = convert("A + B * ( C - D ) * D")
print("Postfix: ", toString(list4))
```

```
Postfix:  AB+
Postfix:  ABC*+
Postfix:  ABC+D+*
Postfix:  ABCD-D**+
```

Systematical Methods:

Conversion of Infix Expressions to Prefix and Postfix

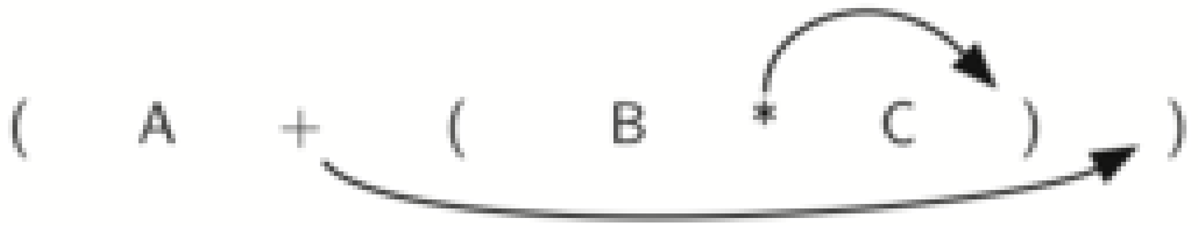
SECTION 19

Fully Parenthesized Expression

- So far, we have used ad hoc methods to convert between infix expressions and the equivalent prefix and postfix expression notations. As you might expect, there are algorithmic ways to perform the conversion that allow any expression of any complexity to be correctly transformed.
- The first technique that we will consider uses the notion of a **fully parenthesized** expression that was discussed earlier. Recall that A + B * C can be written as (A + (B * C)) to show explicitly that the multiplication has precedence over the addition. On closer observation, however, you can see that each parenthesis pair also denotes the beginning and the end of an operand pair with the corresponding operator in the middle.

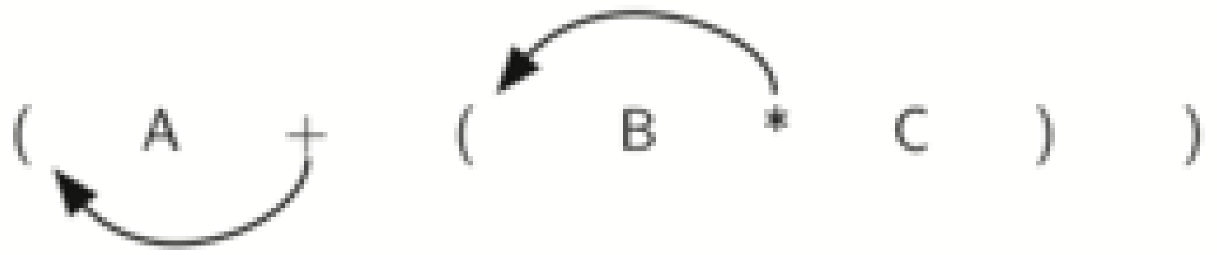
Example
infix to postfix

- Look at the right parenthesis in the subexpression (B * C) above. If we were to move the multiplication symbol to that position and remove the matching left parenthesis, giving us B C *, we would in effect have converted the subexpression to postfix notation.
- If the addition operator were also moved to its corresponding right parenthesis position and the matching left parenthesis were removed, the complete postfix expression would result (see Figure below).



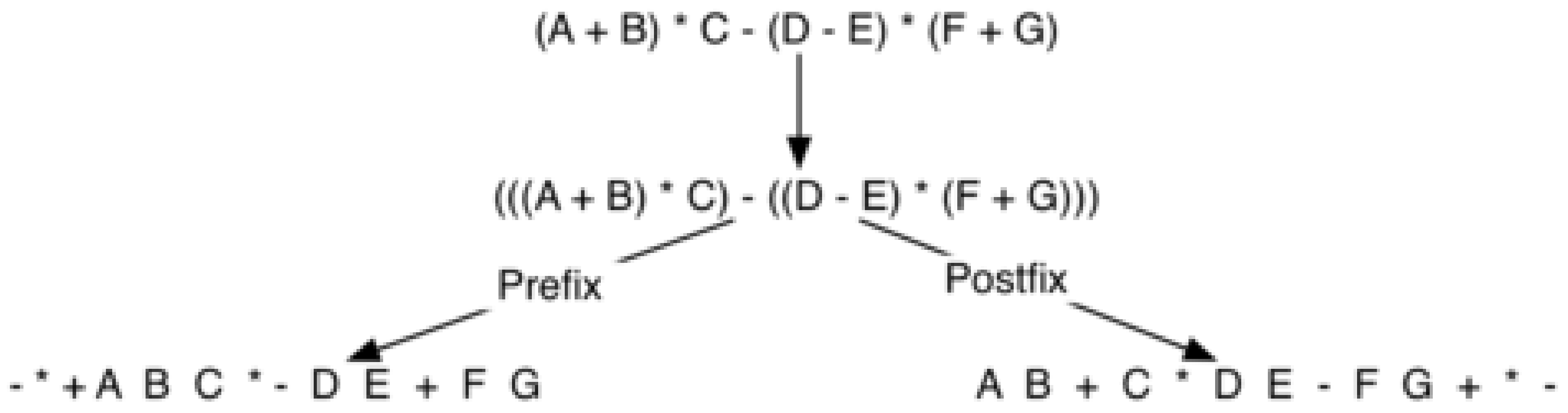
Example
postfix to infix

- If we do the same thing but instead of moving the symbol to the position of the right parenthesis, we move it to the left, we get prefix notation (see Figure below).
- The position of the parenthesis pair is actually a clue to the final position of the enclosed operator.



Systematical Conversion

- So in order to convert an expression, no matter how complex, to either prefix or postfix notation, fully parenthesize the expression using the order of operations. Then move the enclosed operator to the position of either the left or the right parenthesis depending on whether you want prefix or postfix notation.
- Here is a more complex expression: (A + B) * C - (D - E) * (F + G). Figure Below shows the conversion to postfix and prefix notations.



(A + B) * C - (D - E) * (F + G)
(((A + B) * C) - ((D - E) * (F + G)))
Prefix
Postfix
- * + A B C * - D E + F G
A B + C * D E - F G + * -

Methods 1: infix to postfix

SECTION 20

General Infix-to-Postfix Conversion

- Consider once again the expression A + B * C. As shown above, A B C * + is the postfix equivalent.
- We have already noted that the operands A, B, and C stay in their relative positions.
- **It is only the operators that change position.**
- Let’s look again at the operators in the infix expression. The first operator that appears from left to right is +. However, in the postfix expression, + is at the end since the next operator, *, has precedence over addition. The order of the operators in the original expression is reversed in the resulting postfix expression.

Stack is Used for the Storage of Operators

- As we process the expression, the operators have to be saved somewhere since their corresponding right operands are not seen yet. Also, the order of these saved operators may need to be reversed due to their precedence.
- This is the case with the addition and the multiplication in this example.
- **Since the addition operator comes before the multiplication operator and has lower precedence, it needs to appear after the multiplication operator is used.**
- Because of this reversal of order, it makes sense to consider using a stack to keep the operators until they are needed.

Deal With Parentheses

- What about (A + B) * C? Recall that A B + C * is the postfix equivalent. Again, processing this infix expression from left to right, we see + first. In this case, when we see *, + has already been placed in the result expression because it has precedence over * by virtue of the parentheses. We can now start to see how the conversion algorithm will work.
  1. When we see a left parenthesis, we will save it to denote that another operator of high precedence will be coming.
  2. That operator will need to wait until the corresponding right parenthesis appears to denote its position (recall the fully parenthesized technique).
  3. When that right parenthesis does appear, the operator can be popped from the stack.

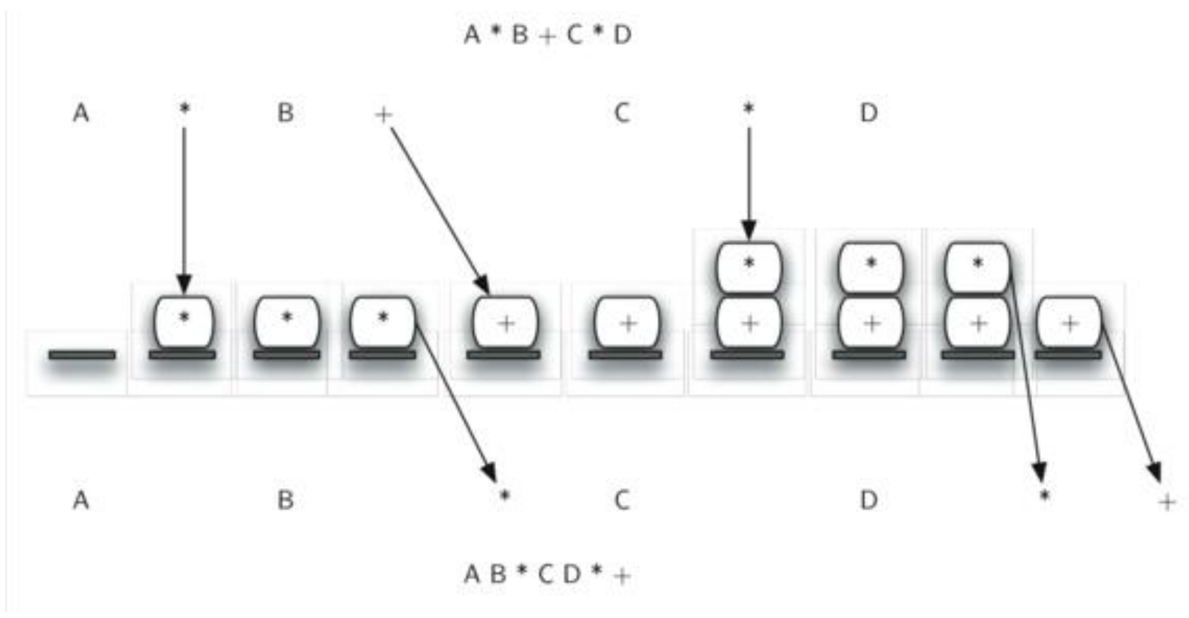
Stack and Parentheses (I)

- As we scan the **infix expression** from left to right, we will use a **stack** to keep the operators.
- This will provide the reversal that we noted in the first example.
- The top of the stack will always be the most recently saved operator.
- Whenever we read a new operator, we will need to consider how that operator compares in precedence with the operators, if any, already on the stack.
- Assume the infix expression is a string of tokens delimited by spaces. The operator tokens are *, /, +, and -, along with the left and right parentheses, (and). The operand tokens are the single-character identifiers A, B, C, and so on.
- The following steps will produce a string of tokens in postfix order. (Next Page)

Stack and Parentheses (II)

1. Create an empty stack called **opstack** for keeping operators. Create an empty list for output.
2. Convert the input infix string to a list by using the string method **split**.
3. 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**.
   - 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.
   - **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.**
4. 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.

Stack and Parentheses (III)



- Figure on the left shows the conversion algorithm working on the expression A * B + C * D. Note that the first * operator is removed upon seeing the + operator. Also, + stays on the stack when the second * occurs, since multiplication has precedence over addition.
- At the end of the infix expression the stack is popped twice, removing both operators and placing + as the last operator in the postfix expression.

Demo Program: InfixToPostfix.py

In order to code the algorithm in Python, we will use a dictionary called prec to hold the precedence values for the operators. This dictionary will map each operator to an integer that can be compared against the precedence levels of other operators (we have arbitrarily used the integers 3, 2, and 1). The left parenthesis will receive the lowest value possible. This way any operator that is compared against it will have higher precedence and will be placed on top of it. Line 15 defines the operands to be any upper-case character or digit. The complete conversion function is shown in ActiveCode 1.

```
class Stack:                                    #InfixToPostfix.py (part 1)
    def __init__(self):
        self.items = []
    def isEmpty(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[len(self.items)-1]
    def size(self):
        return len(self.items)
```

```
def infixToPostfix(infixexpr):                                              # InfixToPostfix.py (part 2)
    prec = {}
    prec["*"] = 3
    prec["/"] = 3
    prec["+"] = 2
    prec["-"] = 2
    prec["("] = 1
    opStack = Stack()
    postfixList = []
    tokenList = infixexpr.split()

    for token in tokenList:
        if token in "ABCDEFGHIJKLMNOPQRSTUVWXYZ" or token in "0123456789":
            postfixList.append(token)
        elif token == '(':
            opStack.push(token)
        elif token == ')':
            topToken = opStack.pop()
            while topToken != '(':
                postfixList.append(topToken)
                topToken = opStack.pop()
        else:
            while (not opStack.isEmpty()) and (prec[opStack.peek()] >= prec[token]):
                postfixList.append(opStack.pop())
            opStack.push(token)

    while not opStack.isEmpty():
        postfixList.append(opStack.pop())
    return " ".join(postfixList)
```

InfixToPostfix.py

InfixToPostfix.py

```python
# InfixToPostfix.py (3)
print(infixToPostfix("A * B + C * D"))
print(infixToPostfix("( A + B ) * C - ( D - E ) * ( F + G )"))
print(infixToPostfix("( A + B ) * ( C + D )"))
print(infixToPostfix("( A + B ) * C"))
print(infixToPostfix("A + B * C"))
```

```
A B * C D * +
A B + C * D E - F G + * -
A B + C D + *
A B + C *
A B C * +
```

Methods 2: Postfix Evaluation

SECTION 21

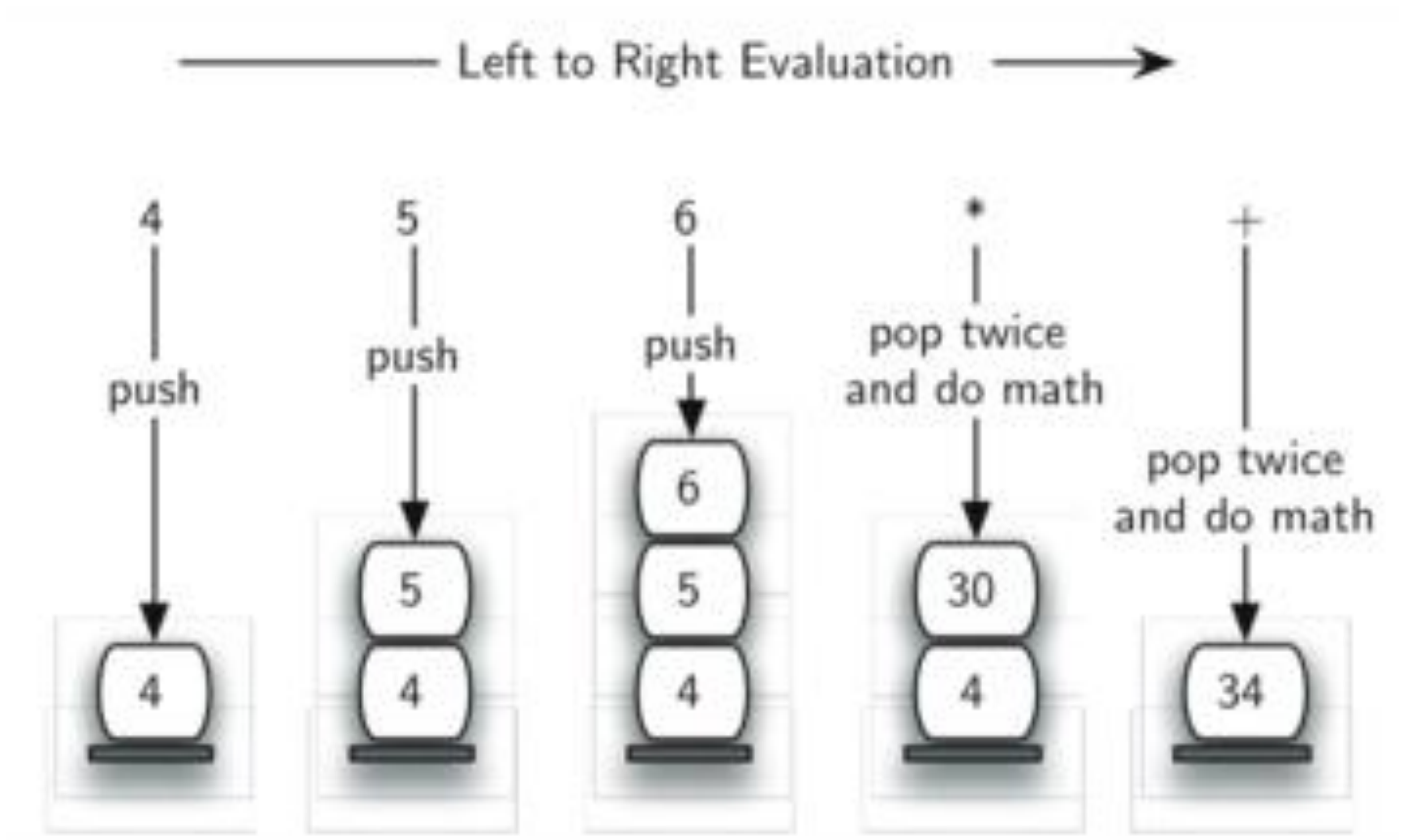
Postfix Evaluation (I)

- As a final stack example, we will consider the evaluation of an expression that is already in postfix notation. In this case, a stack is again the data structure of choice. However, as you scan the postfix expression, it is the operands that must wait, not the operators as in the conversion algorithm above. Another way to think about the solution is that whenever an operator is seen on the input, the two most recent operands will be used in the evaluation.
- To see this in more detail, consider the postfix expression 4 5 6 * +. As you scan the expression from left to right, you first encounter the operands 4 and 5. At this point, you are still unsure what to do with them until you see the next symbol. Placing each on the stack ensures that they are available if an operator comes next.

Postfix Evaluation (II)

- In this case, the next symbol is another operand. So, as before, push it and check the next symbol. Now we see an operator, *. This means that the two most recent operands need to be used in a multiplication operation. By popping the stack twice, we can get the proper operands and then perform the multiplication (in this case getting the result 30).
- We can now handle this result by placing it back on the stack so that it can be used as an operand for the later operators in the expression. When the final operator is processed, there will be only one value left on the stack. Pop and return it as the result of the expression. Figure next page shows the stack contents as this entire example expression is being processed.

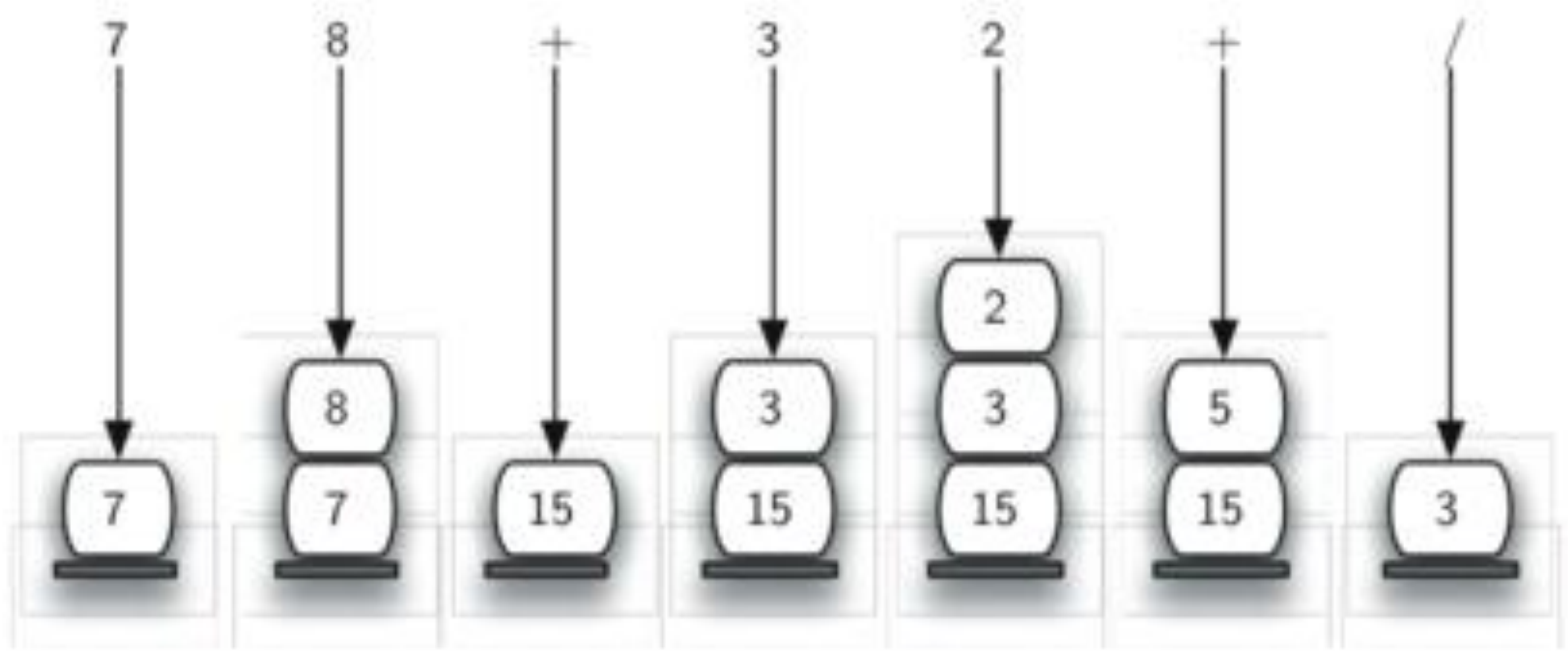
Postfix Evaluation (III)



Postfix Evaluation (IV)

- Figure 11 shows a slightly more complex example, 7 8 + 3 2 + /.
- There are two things to note in this example.
  - First, the stack size grows, shrinks, and then grows again as the subexpressions are evaluated.
  - Second, the division operation needs to be handled carefully.
- Recall that the operands in the postfix expression are in their original order since postfix changes only the placement of operators.
- When the operands for the division are popped from the stack, they are reversed. Since division is not a commutative operator, in other words 15/5 is not the same as 5/15, we must be sure that the order of the operands is not switched.

Postfix Evaluation (V)



Algorithm for Evaluation of Expressions

Assume the postfix expression is a string of tokens delimited by spaces. The operators are *, /, +, and - and the operands are assumed to be single-digit integer values. The output will be an integer result.

1. Create an empty stack called operandStack.
2. Convert the string to a list by using the string method split.
3. Scan the token list from left to right.
   - If the token is an operand, convert it from a string to an integer and push the value onto the operandStack.
   - If the token is an operator, *, /, +, or -, it will need two operands. Pop the operandStack twice. 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 operandStack.
4. When the input expression has been completely processed, the result is on the stack. Pop the operandStack and return the value.

Demo Program: PostfixEval.py

- The complete function for the evaluation of postfix expressions is shown in PostfixEval.py.
- To assist with the arithmetic, a helper function doMath is defined that will take two operands and an operator and then perform the proper arithmetic operation.

```python
def postfixEval(postfixExpr):
    operandStack = Stack()
    tokenList = postfixExpr.split()

    for token in tokenList:
        if token in "0123456789":
            operandStack.push(int(token))
        else:
            operand2 = operandStack.pop()
            operand1 = operandStack.pop()
            result = doMath(token,operand1,operand2)
            operandStack.push(result)
    return operandStack.pop()
```

```
def doMath(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 + /'))
```

```
C:\Python\Python36\python.exe "C:/Eric_Chou/Python Course/Python Intermediate
Programming/PyDev/Unit3/Examples/postfixEval.py"
3.0

Process finished with exit code 0
```

Postfix to Infix Conversion

SECTION 22

Conversion of a Postfix Expression into an Infix Expression

Step 1: Read the postfix expression from left to right .

Step 2: If the symbol is an operand, then insert it onto the front of a queue.

Step 3: If the symbol is an operator, pop two symbols from the front of the queue and create it as a string by placing the operator in between the operands and insert the resulted string to the front of the queue.

Step 4: Repeat steps 2 and 3 till the end of the postfix expression.

Step 5: EXIT

Example

- Consider the following Prefix Expression...

  A B - C D * +

- The above prefix expression can be converted into infix expression using Stack data Structure as follows...

- The equivalent infix Expression is as follows...

  ((A - B) + (C * D))

Reading Character	Infix Expression		
A	A		
B	B	A	
-	(A-B)		
C	C	(A-B)	
D	D	C	(A-B)
*	(C*D)	(A-B)	
+	(A-B)+(C*D)		

PostfixToInfix.py

```python
def isOperand(x):
    return ((x >= 'a' and x <= 'z') or
            (x >= 'A' and x <= 'Z'))

def postfixToInfix(postfix) :
    s = []
    for i in range(len(postfix)):
        # Insert at head
        if (isOperand(postfix[i])) :
            s.insert(0, postfix[i])
        else:
            op1 = s[0]
            s.pop(0)
            op2 = s[0]
            s.pop(0)
            s.insert(0, "(" + op2 + postfix[i] + op1 + ")")
    return s[0]

print(postfixToInfix("ab-c+de*-"))
```

(((a-b)+c)-(d*e))

Prefix to Infix Conversion

SECTION 23

Conversion of a Prefix Expression into an Infix Expression

Step 1: Read the prefix expression from right to left .

Step 2: If the symbol is an operand, then push it onto the stack.

Step 3: If the symbol is an operator,

pop two symbols from the stack and create it as a string by placing the operator in between the operands and push the resulted string back to stack.

Step 4: Repeat steps 2 and 3 till the end of the prefix expression.

Step 5: EXIT

Example

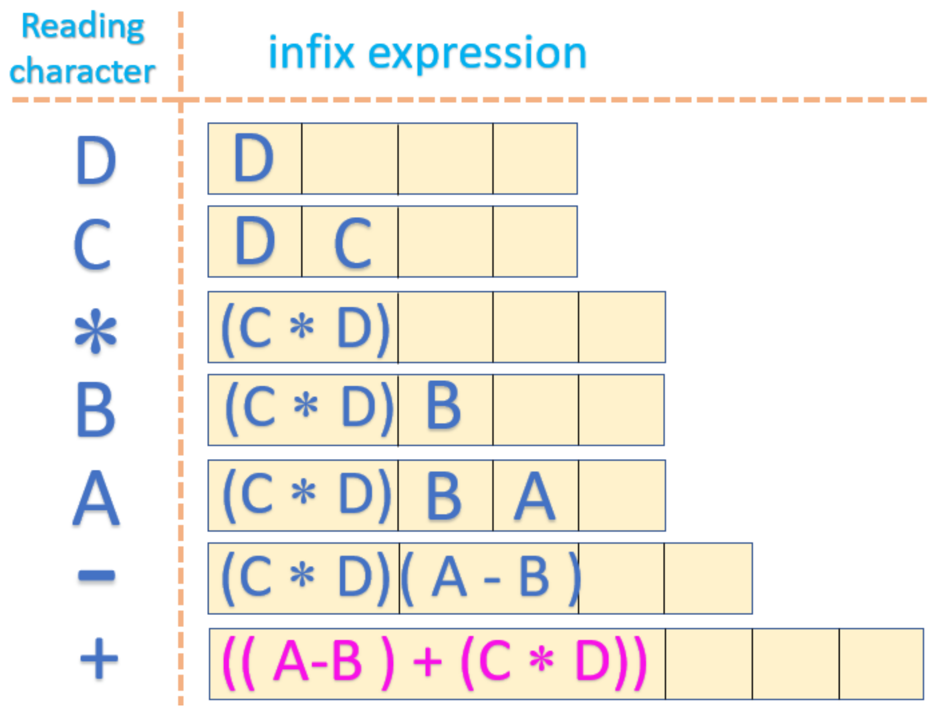
- Consider the following Prefix Expression...

  + - A B * C D

- The above prefix expression can be converted into infix expression using Stack data Structure as follows...

- The equivalent infix Expression is as follows...

  ((A - B) + (C * D))



Reading character	infix expression			
D	D			
C	D	C		
*	(C * D)			
B	(C * D)	B		
A	(C * D)	B	A	
-	(C * D)	(A - B)		
+	((A-B) + (C * D))			

PrefixToInfix.py

```python
from Stack import Stack
def prefixToInfix(prefix):
    stack = Stack()
    # read prefix in reverse order
    i = len(prefix) - 1
    while i >= 0:
        if not isOperator(prefix[i]):
            # symbol is operand
            stack.push(prefix[i])
            i -= 1
        else:
            # symbol is operator
            str = "(" + stack.pop() + prefix[i] + stack.pop() + ")"
            stack.push(str)
            i -= 1

    return stack.pop()

def isOperator(c):
    return c in "*+-/^()"

print(prefixToInfix("*-A/BC-/AKL"))
```

```
((A-(B/C))*((A/K)-L))
```

Prefix to Postfix Conversion

SECTION 24

Conversion of a Prefix Expression into an Postfix Expression

Step 1: Read the prefix expression from right to left .

Step 2: If the symbol is an operand, then push it onto the stack.

Step 3: If the symbol is an operator,

pop two symbols from the stack and create it as a string by placing the operator after the operands and push the resulted string back to stack.

Step 4: Repeat steps 2 and 3 till the end of the prefix expression.

Step 5: EXIT

Example

- Consider the following Prefix Expression...

  + - A B * C D

- The above prefix expression can be converted into postfix expression using Stack data Structure as follows...

- The equivalent postfix Expression is as follows...

  A B - C D * +

Reading character	postfix expression			
D	D			
C	D	C		
*	C D*			
B	C D*	B		
A	C D*	B	A	
-	C D*	A B-		
+	A B - C D* +			

PrefixToPostfix.py

```python
def prefixToPostfix(prefix):
    # Reversing the order
    s = prefix[::-1]
    stack = []

    for i in s:
        if i in "+-*/^":
            a = stack.pop()
            b = stack.pop()
            temp = a+b+i
            stack.append(temp)
        else:
            stack.append(i)

    p = "".join(stack)
    return p

print(prefixToPostfix("*-A/BC-/AKL"))
```

```
ABC/-AK/L-*
```

Summary

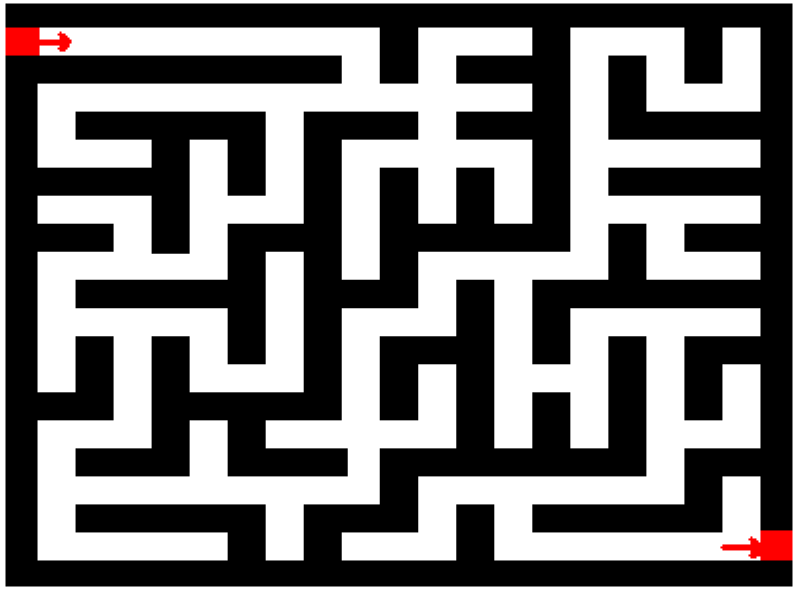
SECTION 25

Summary

- It is important to note that in both the postfix conversion and the postfix evaluation programs we assumed that there were no errors in the input expression.
- Using these programs as a starting point, you can easily see how error detection and reporting can be included. We leave this as an exercise at the end of the chapter.

Applications

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.

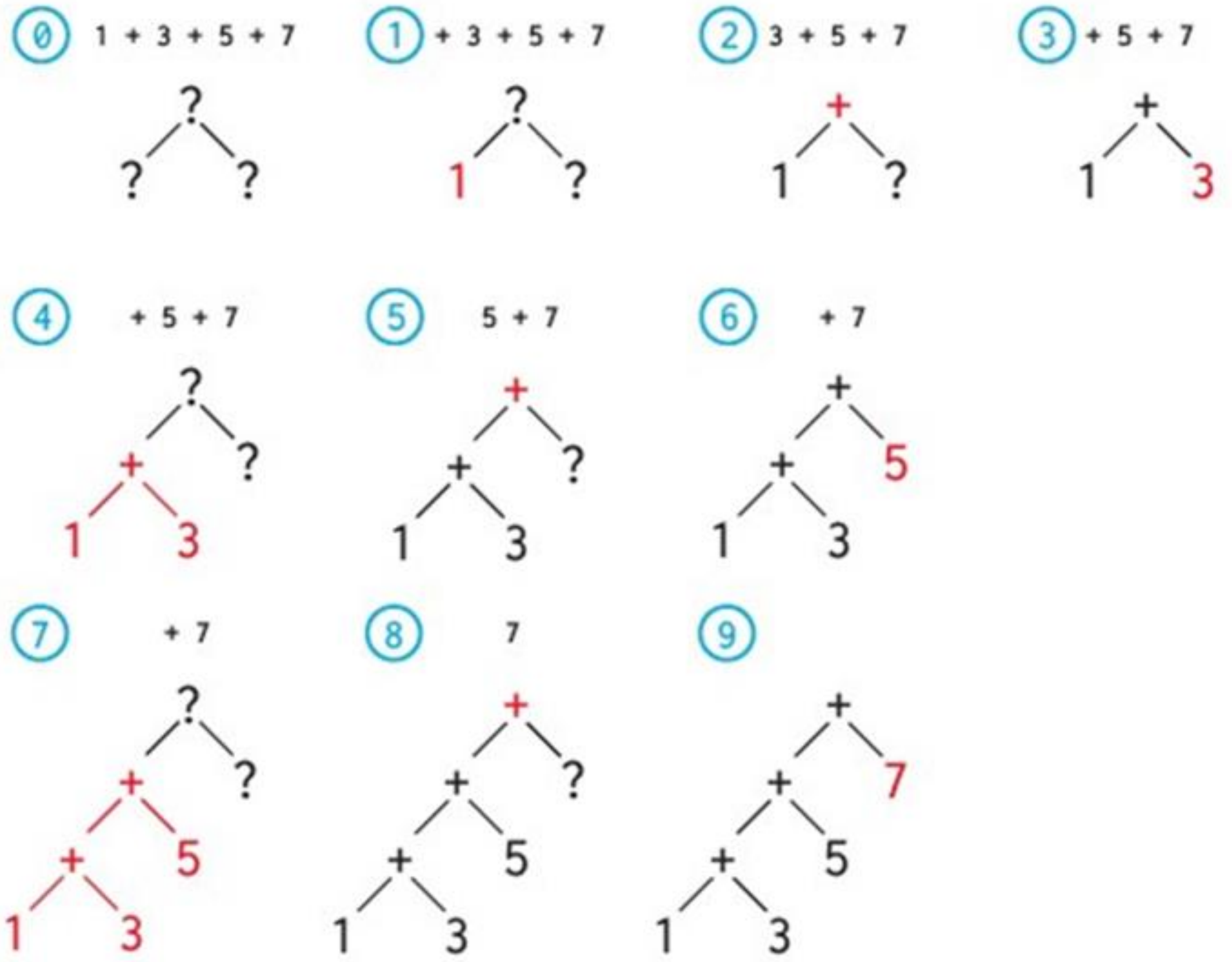


Backtracking. This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit? Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.

- Language processing:
    - space for parameters and local variables is created internally using a stack.
    - compiler's syntax check for matching braces is implemented by using stack.
    - support for recursion

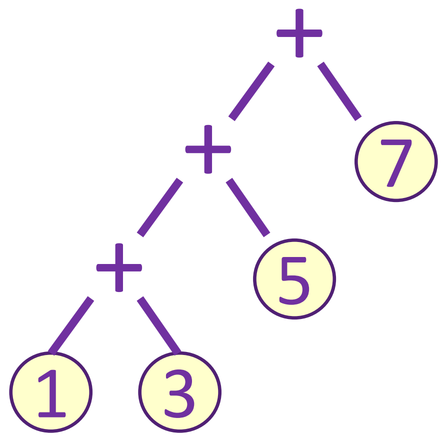
Syntax Tree, Stack, and Expression Evaluation

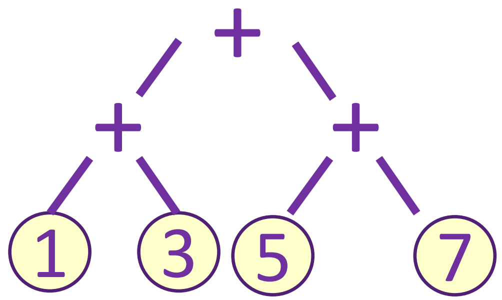
- In this lecture, we briefly go over functional programming and its relationship with recursion.
- Then, we work on the tree construction and various type of trees.
- After that, we talk about parsing tree.
- From an expression to a evaluation tree using stack and the different expression representation conversion and evaluation.
- More detailed syntax tree evaluation will be covered in the parsing and the interpreter design chapters.

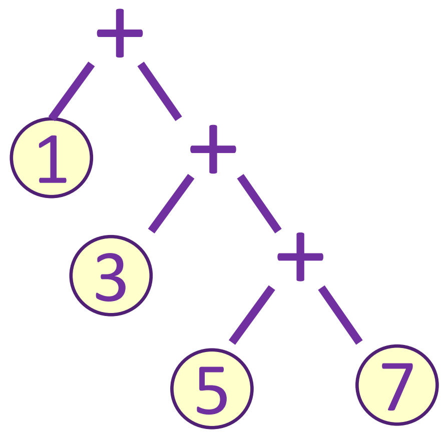


0 1 + 3 + 5 + 7
1 + 3 + 5 + 7
2 3 + 5 + 7
3 + 5 + 7
4 + 5 + 7
5 5 + 7
6 + 7
7 + 7
8 7
9

Ambiguity







End of Chapter 6B