.UBAfiuba
FACULTAD DE INGENIERÍA

# Recursividad y Manejo de Memoria Dinámica

*Pilas, heaps, garbage collection y gramáticas formales*

# Hoja de ruta de la clase

1. **¿Por qué recursividad?**
   *Estructuras auto-similares y problemas que piden recursión.*
2. **Anatomía de un algoritmo recursivo**
   *Caso base + caso recursivo, aplicado con metodología de Pólya.*
3. **Pila de ejecución y árbol de recursión**
   *Dos vistas del costo: profundidad de pila y cantidad de nodos.*
4. **Iteración vs. recursión**
   *Tres criterios para elegir entre las dos formas.*
5. **Patrones recursivos canónicos**
   *Suma, potencia rápida, búsqueda binaria recursiva.*
6. **Recursividad de cola y la política de Python**
   *Ideal teórico no aplicado; razones del lenguaje.*
7. **RecursionError y los límites de la pila**
   *Dos capas de protección: el intérprete y el SO.*
8. **El heap y los tres pecados capitales**
   *Memoria dinámica en C; leaks, dangling pointers, double free.*
9. **El recolector de basura de Python**
   *Reference counting + detección generacional de ciclos.*
10. **[Holístico] Gramáticas formales**
    *BNF, EBNF, árboles de derivación, parsers de descenso recursivo.*

1

# ¿Por qué recursividad?

*Las estructuras auto-similares y los problemas que se descomponen en versiones más pequeñas de sí mismos son escenarios naturales de la recursión.*

# Estructuras auto-similares en lo cotidiano

*Estos son problemas donde lo grande contiene una versión más pequeña de sí mismo. La forma natural de definirlos —y de resolverlos— es recursiva.*

### Carpetas anidadas

Una carpeta puede contener otras carpetas. Para listar todos los archivos hace falta entrar en cada una y, dentro, repetir el procedimiento.

*El comando ls -R es exactamente este patrón.*

### Definiciones matemáticas

Factorial: $n! = n \cdot (n-1)!$ con $0! = 1$

Fibonacci: $F(n) = F(n-1) + F(n-2)$

*La definición ES recursiva. Forzar iteración pierde claridad.*

### Sintaxis de lenguajes

Una expresión es:

un número, o

una expresión + un término, o

( una expresión )

*Las gramáticas formales se definen recursivamente (Sección 10).*

2

# Anatomía de un algoritmo recursivo

*Toda función recursiva tiene dos ingredientes: el caso base que detiene la recursión, y el caso recursivo que invoca a la propia función con una entrada estrictamente más pequeña.*

# Estructura general (con metodología de Pólya)

## Las cuatro fases

1. **Análisis**

   *Identificar qué entrada es la más simple posible (caso base) y cómo el problema se descompone en sub-problemas más pequeños.*

2. **Diseño**

   *Escribir la recurrencia: f(n) en términos de f(k) con k < n. Verificar que toda recurrencia eventualmente alcanza el caso base.*

3. **Codificación**

   *Caso base como primer bloque del cuerpo. Caso recursivo después. Documentar la precondición y la postcondición.*

4. **Evaluación**

   *Trazar a mano para n pequeño (n=3). Verificar el árbol de recursión. Estimar profundidad de pila.*

## Esqueleto del código

```
Algoritmo f(entrada):
    // Sección declarativa (docstring)
    // Pre: <condición sobre la entrada>
    // Pos: <qué calcula la función>

    si <caso base>:
        retornar <valor directo>
    sino:
        // Combinar resultado(s)
        // de subproblema(s) más pequeño(s)

        retornar <expresión con f(...)>
```

# Ejemplo canónico: factorial

### Pseudocódigo

```
Algoritmo factorial(n):
    // Pre: n >= 0 entero
    // Pos: retorna n!
    si n = 0:
        retornar 1
    sino:
        retornar n * factorial(n - 1)
```

### Python

```python
def factorial(n):
    """Calcula n! recursivamente."""

    if n == 0:
        return 1

    return n * factorial(n - 1)
```

🔗 **Caso base como primer bloque del cuerpo - convención del curso**

*Poner el caso base antes del caso recursivo no es estética: cuando el caso base falta o se evalúa después, la recursión nunca se detiene y termina en RecursionError.*

*La disciplina de "caso base primero" es la causa de prevención más simple y efectiva.*

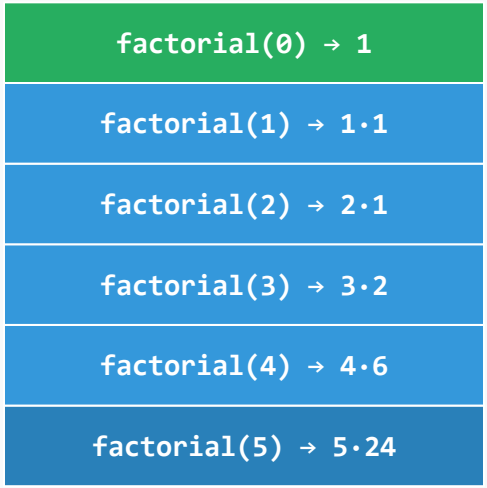
3

# Pila de ejecución y árbol de recursión

*Dos vistas del costo de un algoritmo recursivo: la pila mide la profundidad del camino actual; el árbol mide el total de llamadas.*

# La pila durante factorial(5)

*Cada llamada recursiva agrega un frame en la pila. Cuando se alcanza el caso base, los frames se destruyen en orden inverso (LIFO), retornando los valores hacia arriba.*




*tope (caso base)*

*crece ↑*

*base de la pila*

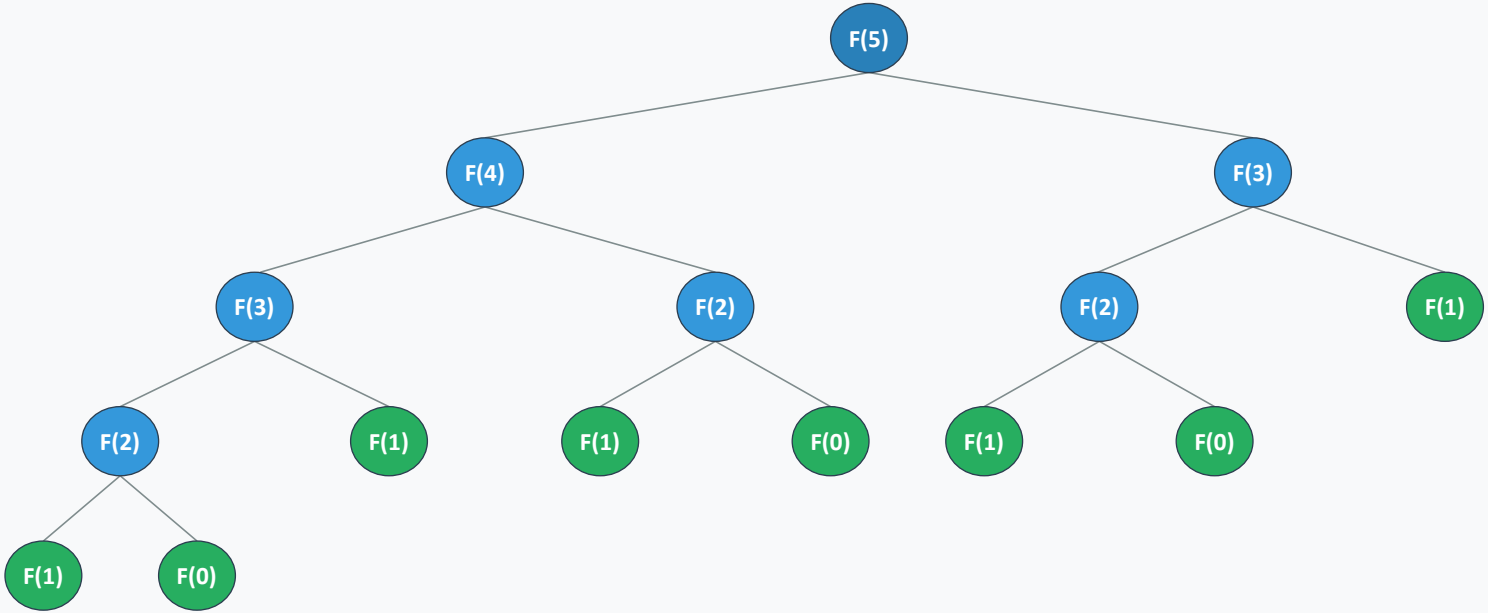
## Traza de retornos

```
factorial(0) = 1
factorial(1) = 1 · 1   = 1
factorial(2) = 2 · 1   = 2
factorial(3) = 3 · 2   = 6
factorial(4) = 4 · 6   = 24
factorial(5) = 5 · 24  = 120
```

**n + 1**
frames máximos en pila

**O(n)**
tiempo total

# Árbol de recursión: Fibonacci(5) recomputa subproblemas




*Total: 15 llamadas para calcular F(5). F(2) se calcula 3 veces; F(1) se calcula 5 veces; F(0) se calcula 3 veces.*

🔗 **Recomputación masiva → costo exponencial**

*La memoización (Sem. 13) es la cura.*

4

# Iteración vs. recursión

*No es una elección ideológica. Es una elección que se justifica articulando tres criterios: legibilidad, profundidad de pila, eficiencia.*

# Tabla comparativa

| Criterio | Iteración | Recursión |
|---|---|---|
| Legibilidad | Buena para problemas planos: bucles sobre listas, acumuladores, conteos. | Buena para problemas auto-similares: árboles, divide-y-vencerás, recurrencias matemáticas directas. |
| Uso de pila | O(1) frames - no crece con la entrada. | O(profundidad de la recursión) frames - puede ser O(log n), O(n) o peor. |
| Eficiencia | Sin overhead de creación/destrucción de frames; el bucle se compila directo a saltos. | Pequeño overhead por frame, en general invisible. Si el árbol tiene subproblemas repetidos, el costo asintótico empeora drásticamente. |

🔗 **Regla práctica del curso**

*Si el problema admite una formulación natural con un acumulador, elegir iteración. Si la estructura del problema o de los datos es jerárquica/auto-similar, elegir recursión. Para todo lo demás, las dos formas son razonables: elegir la que se lea más clara.*

# Cuándo conviene cada forma

## → Recursión natural

**Los datos tienen estructura jerárquica.**

*Ejemplos:*

- Árboles, JSONs anidados
- Sistemas de archivos
- Expresiones aritméticas
- Torres de Hanói

*El árbol de llamadas refleja la estructura del problema. Iterar requeriría una pila explícita en el código del usuario.*

## ↔ Las dos funcionan

**Cualquiera de las dos formas es razonable; elegir por claridad.**

*Ejemplos:*

- Búsqueda binaria
- Cálculo de potencias
- MCD (Euclides)
- Conversión de bases

*Profundidad logarítmica: la pila no es problema. Diferencia de eficiencia menor al 10-30%.*

## → Iteración natural

**Recursión lineal sin valor adicional.**

*Ejemplos:*

- Suma/conteo de lista
- Recorrer una secuencia
- Cálculo factorial (!)
- Procesar archivo línea a línea

*Profundidad lineal: para n grande llega a RecursionError. La iteración no tiene este problema.*

5

# Patrones recursivos canónicos

*Tres ejemplos esquemáticos: recursión lineal, divide y vencerás con dos llamadas, divide y vencerás con una llamada (cola).*

# Patrón 1 - Recursión lineal: suma de lista

*Una llamada por nivel; el árbol degenera en una cadena de profundidad n. Útil para entender el modelo, no para usar en producción.*

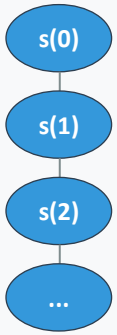
```python
def suma_rec(lista, i=0):
    """Suma desde la posición i hasta el final."""

    if i == len(lista):
        return 0 # caso base

    return lista[i] + suma_rec(lista, i + 1)

# Para una lista de tamaño n, la pila
# tiene n+1 frames simultáneos al llegar
# al caso base. Para n = 10**5: RecursionError.
```

**Estructura del árbol**




*Cada nodo invoca exactamente 1 hijo: un árbol que es una cadena.*

**O(n)**

profundidad de pila lineal: una llamada por nivel

# Patrón 2 - Divide y vencerás: potencia rápida

*Cada llamada reduce el problema a la mitad. El árbol tiene profundidad $log_2(n)$ y solo $log_2(n)$ nodos: una de las recursiones más eficientes posibles.*

```python
def potencia(base, n):
    """Calcula base**n con O(log n) multiplicaciones."""

    if n == 0:
        return 1

    mitad = potencia(base, n >> 1)

    if n & 1:                   # n impar
        return mitad * mitad * base
    return mitad * mitad
```

### Eficiencia para potencia(2, n)

| n | Ingenuo | Rápido |
|---|---|---|
| 10 | 10 | 4 |
| 100 | 100 | 7 |
| $10^6$ | $10^6$ | 20 |
| $10^9$ | $10^9$ | 30 |

🔗 **Operador n >> 1 - conexión con la ALU (Semana 1)**

*Dividir un entero positivo por 2 es desplazar todos sus bits una posición a la derecha. Esta elección refleja la operación a nivel de la ALU; en bytecode CPython, n >> 1 y n // 2 son equivalentes.*

# Patrón 3 - Búsqueda binaria recursiva (Semana 5 revisitada)

*La búsqueda binaria fue iterativa en la Semana 5. Su definición natural es recursiva: comparar con el medio y, si no es igual, buscar en una de las dos mitades.*

```python
def buscar_bin_rec(lista, valor, izq, der):
    """Devuelve índice de valor o -1 si no está."""

    if izq > der: # caso base
        return -1

    medio = (izq + der) >> 1

    if lista[medio] == valor:
        return medio

    if lista[medio] < valor:
        return buscar_bin_rec(lista, valor, medio + 1, der)

    return buscar_bin_rec(lista, valor, izq, medio - 1)
```

## Recursiva vs iterativa

### Eficiencia

Ambas son O(log n) en tiempo.

La iterativa: O(1) en pila.

*La recursiva: O(log n) en pila — irrelevante en la práctica.*

### Cuándo usar cuál

Para n hasta $10^9$, $\log_2(n) \approx 30$ frames: la pila no es problema.

*La elección es por legibilidad: la versión recursiva es más cercana a la formulación matemática del divide y vencerás.*

6

# Recursividad de cola y la política de Python

*Cuando la llamada recursiva es lo último que hace la función, podría optimizarse para no consumir pila. Python deliberadamente no lo hace.*

# Recursividad de cola (Tail Call Optimization)

*Llamada de cola: la última operación de la función es la llamada recursiva, sin operaciones pendientes después. Algunos compiladores la transforman en un salto, eliminando el frame.*

**NO es recursión de cola**

```python
def factorial(n):
    if n == 0:
        return 1
    return n * factorial(n - 1)
```

↑ multiplicar es posterior a la llamada

**SÍ es recursión de cola**

```python
def factorial_tco(n, acum=1):
    if n == 0:
        return acum
    return factorial_tco(n - 1, n * acum)
```

↑ nada después de la llamada

🔗 **Lenguajes que SÍ aplican TCO**

*Funcionales puros (Scheme, ML, Haskell): la garantizan. Compilados de bajo nivel (C, C++, Rust con -O2): la aplican. Java: parcial.*
***Python:** no la aplica, ni la aplicará. La razón es deliberada de diseño y se discute en la siguiente diapositiva.*

# ¿Por qué Python rechaza la TCO?

### Tracebacks legibles

Si los frames intermedios desaparecieran, el traceback al lanzar una excepción mostraría solo la última llamada y no el camino completo. La TCO destruiría una herramienta de depuración fundamental.

### Coherencia con la introspección

El módulo inspect, los profilers y los debuggers asumen que cada llamada deja un frame visible. Cambiar esto rompería un ecosistema entero de herramientas.

### Filosofía no-funcional

Python es predominantemente imperativo. La iteración con for y while es idiomática y eficiente; usar recursión de cola para simular un bucle sería una práctica forzada y poco pythonista.

7

# RecursionError y los límites de la pila

*Python protege a sus programas de un stack overflow del SO con un límite artificial controlado por el intérprete. Dos capas de protección actuando en concierto.*

# RecursionError: protección suave del intérprete

*Cuando la profundidad excede sys.getrecursionlimit() (por defecto 1000), el intérprete levanta un RecursionError manejable.*

```
import sys
sys.getrecursionlimit()       # 1000 (default)

def f(n): return f(n + 1)

try:
    f(0)
except RecursionError as e:
    print('capturado')
```

### Las dos capas de protección

**Capa 1: el intérprete**

Cuenta sus propios frames y aborta con RecursionError.

*Excepción capturable, programa sigue.*

**Capa 2: el SO**

Si la pila se desborda físicamente: SIGSEGV.

*Crash del proceso, sin captura posible.*

***setrecursionlimit() levanta solo la capa 1, no la capa 2.***

8

# El heap revisitado: memoria dinámica

*Hasta aquí estudiamos la pila. Ahora le toca al heap: la región de memoria dinámica donde viven los objetos cuyo tamaño o tiempo de vida no se conoce en compilación.*

# El modelo de memoria del proceso (Semana 7 revisitada)

| | | |
|---|---|---|
| **PILA** | ↓ *crece* | Frames de funciones. Tamaño fijo conocido en compilación. LIFO automática. |
| **(libre)** | | Espacio entre pila y heap, disponible para que ambas crezcan. |
| **HEAP** | ↑ *crece* | Objetos con tamaño determinado en ejecución. Asignación / liberación explícita o asistida. |
| **DATOS** | | Variables globales y estáticas. Tamaño fijo conocido en compilación. |
| **TEXTO** | | Código ejecutable (read-only). Cargado por el SO al iniciar el proceso. |

🔗 **Conexión con Semana 8: la lista de bloques libres**

*El heap se gestiona internamente con una free list, idéntica al patrón de bloques libres del sistema de archivos visto la semana pasada.*

# Los tres pecados capitales del manejo manual (C/C++)

*La libertad de malloc/free tiene tres errores clásicos. Todos comparten una característica perniciosa: el momento del error y el momento del síntoma están separados.*

## Memory leak

**malloc sin free correspondiente.**

El proceso retiene memoria indefinidamente. En un bucle, el consumo crece monótonamente hasta agotar la memoria.

*Síntoma: programa lento horas después; OOM-killer del kernel mata al proceso.*

## Dangling pointer

**Puntero a memoria ya liberada.**

Use-after-free: leer o escribir por ese puntero es comportamiento indefinido. La memoria puede haber sido reasignada a otro objeto.

*Síntoma: corrupción silenciosa de datos no relacionados; crash impredecible.*

## Double free

**free dos veces sobre el mismo puntero.**

Corrompe la free list interna del gestor del heap. La próxima asignación devuelve memoria inválida o falla.

*Síntoma: crash en una operación de memoria posterior, sin relación aparente con la causa.*

### 🔗 Vulnerabilidades CVE

*Una porción significativa de las vulnerabilidades reportadas en el catálogo CVE de MITRE se origina en estos tres errores.*

9

# El recolector de basura de Python

*Python toma una decisión radicalmente distinta a la de C: el programador nunca llama a free. La memoria se libera automáticamente.*

# Conteo de referencias (reference counting)

*Cada objeto lleva un contador interno: cuántas referencias actualmente lo apuntan. Cuando llega a 0, el objeto se libera inmediatamente.*

```
import sys

x = [1, 2, 3]
sys.getrefcount(x)        # 2 (x + arg)

y = x                     # nueva ref
sys.getrefcount(x)        # 3

del x                     # ref menos
sys.getrefcount(y)        # 2

del y                     # contador → 0
                          # objeto liberado
```

## Tres virtudes

### Determinismo

*Liberación inmediata al perder la última referencia.*

### Localidad

*Libera mientras el objeto aún está en caché de CPU.*

### Simplicidad

*Algoritmo trivial: ++/-- y comparar con 0.*

# El problema de los ciclos

*Dos objetos que se apuntan mutuamente: ningún refcount llega a 0, pero nadie puede acceder a ellos. Forman una isla de basura.*

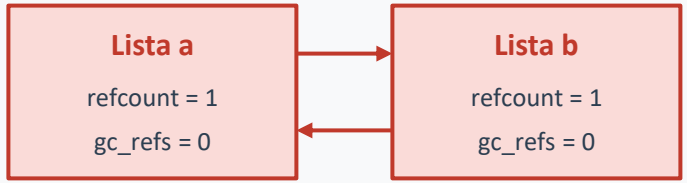
```
a = []
b = []

a.append(b)            # a contiene b
b.append(a)            # b contiene a

del a
del b

# Ambos refcount = 1 (apuntados # por el otro),
pero nadie puede acceder a ellos: isla de basura.
```

### Diagrama del ciclo




*Ningún gc_refs > 0 y ningún acceso desde una raíz: el detector de ciclos los liberará.*

### 🔗 La solución: detector generacional de ciclos

*CPython recorre periódicamente los objetos contenedores buscando grupos cuyas referencias internas se cancelan. Hipótesis generacional: los objetos jóvenes se revisan más seguido que los viejos. Tres generaciones (0, 1, 2) con umbrales (700, 10, 10).*

# Tres estrategias de gestión del heap

| Aspecto | C/C++ manual | Python (refcount + GC) | Java/Go (GC tracing) |
|---|---|---|---|
| Liberación | Explícita: free() | Automática: refcount o ciclo | Automática: pasadas periódicas |
| Determinismo | Total (decisión del programador) | Alto (refcount es inmediato) | Bajo (GC corre cuando le conviene) |
| Memory leaks | Posibles (olvido de free) | Solo en ciclos no detectados | Casi imposibles |
| Dangling pointers | Posibles | Imposibles por diseño | Imposibles por diseño |
| Pausas (stop-the-world) | Ninguna | Cortas (en detección de ciclos) | Variables, optimizables |

10

# Gramáticas formales y árboles de derivación

*[Holístico] La recursión no es solo una técnica de programación: es la herramienta conceptual fundamental para describir lenguajes.*

# Notación BNF/EBNF — la sintaxis se define recursivamente

*Una gramática libre de contexto especifica qué cadenas pertenecen a un lenguaje. Casi todos los lenguajes de programación se especifican con CFGs en notación EBNF.*

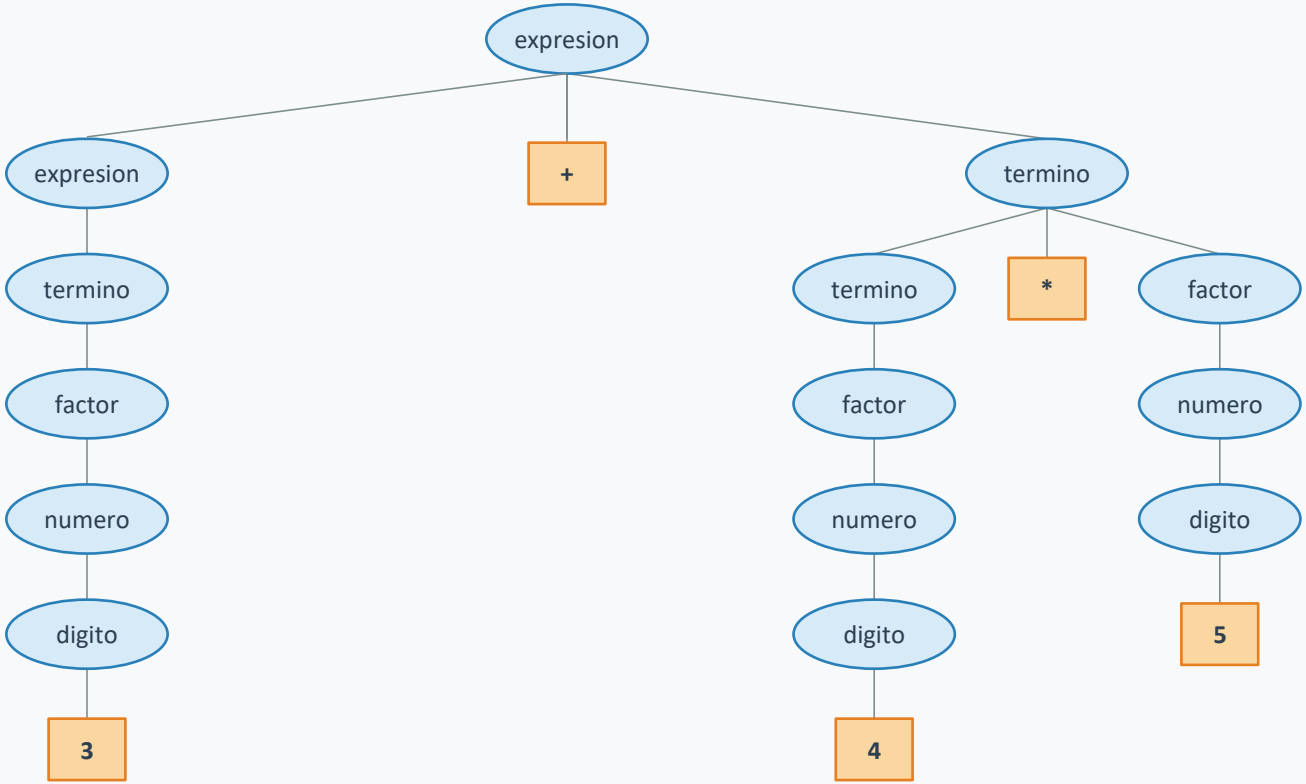
**Gramática EBNF para expresiones aritméticas**

```
expresion ::= termino { ( + | - ) termino }
termino   ::= factor { ( * | / ) factor }
factor    ::= numero   |  ( expresion )
numero    ::= digito { digito }
```

🔗 **La definición ES recursiva**

*<expresion> se refiere a sí misma; <termino> también; <factor> contiene <expresion> entre paréntesis. La estructura recursiva de la gramática es lo que permite anidar paréntesis arbitrariamente. La precedencia de operadores no es una decisión separada: es la jerarquía de la gramática.*

# Árbol de derivación para "3 + 4 * 5"




*La estructura del árbol revela que la multiplicación se evalúa primero (es subárbol del operando derecho de +), porque la gramática define <termino> dentro de <expresion>: la jerarquía gramatical determina la precedencia.*

# Parser de descenso recursivo: el código ES la gramática

*Una función por cada no-terminal de la gramática. Las llamadas mutuamente recursivas reflejan exactamente las producciones EBNF.*

```python
def parse_expresion(tokens, i):
    valor, i = parse_termino(tokens, i)
    while i < len(tokens) and tokens[i] in ('+', '-'):
        op = tokens[i]
        derecha, i = parse_termino(tokens, i + 1)
        valor = valor + derecha if op == '+' else valor - derecha
    return valor, i

def parse_factor(tokens, i):
    if tokens[i] == '(':
        valor, i = parse_expresion(tokens, i + 1)
        return valor, i + 1
    return int(tokens[i]), i + 1
```

🔗 **CPython usa un parser de descenso recursivo (PEG, desde 3.9)**

*Cada vez que ejecutamos código Python, la primera operación del intérprete es invocar este parser recursivo sobre el código fuente.*

# La pila vuelve a aparecer

*Una idea unificadora atraviesa todo lo visto.*

| | |
|---|---|
| **Semana 4** | La pila como mecanismo del SO para gestionar llamadas a funciones. |
| **Semana 7** | La pila como uno de los cuatro segmentos del modelo de memoria del proceso. |
| **Semana 9 (hoy)** | La pila como el escenario donde la recursividad se materializa. |
| **Semana 9 (Sec. 10)** | La pila como el objeto matemático que define qué lenguajes son CFG (autómatas de pila). |

***La pila no es solo cómo se implementa la recursividad: es lo que la recursividad es.***

*Próxima clase (Semana 10): Estructuras de datos — listas, tuplas, diccionarios y conjuntos*