

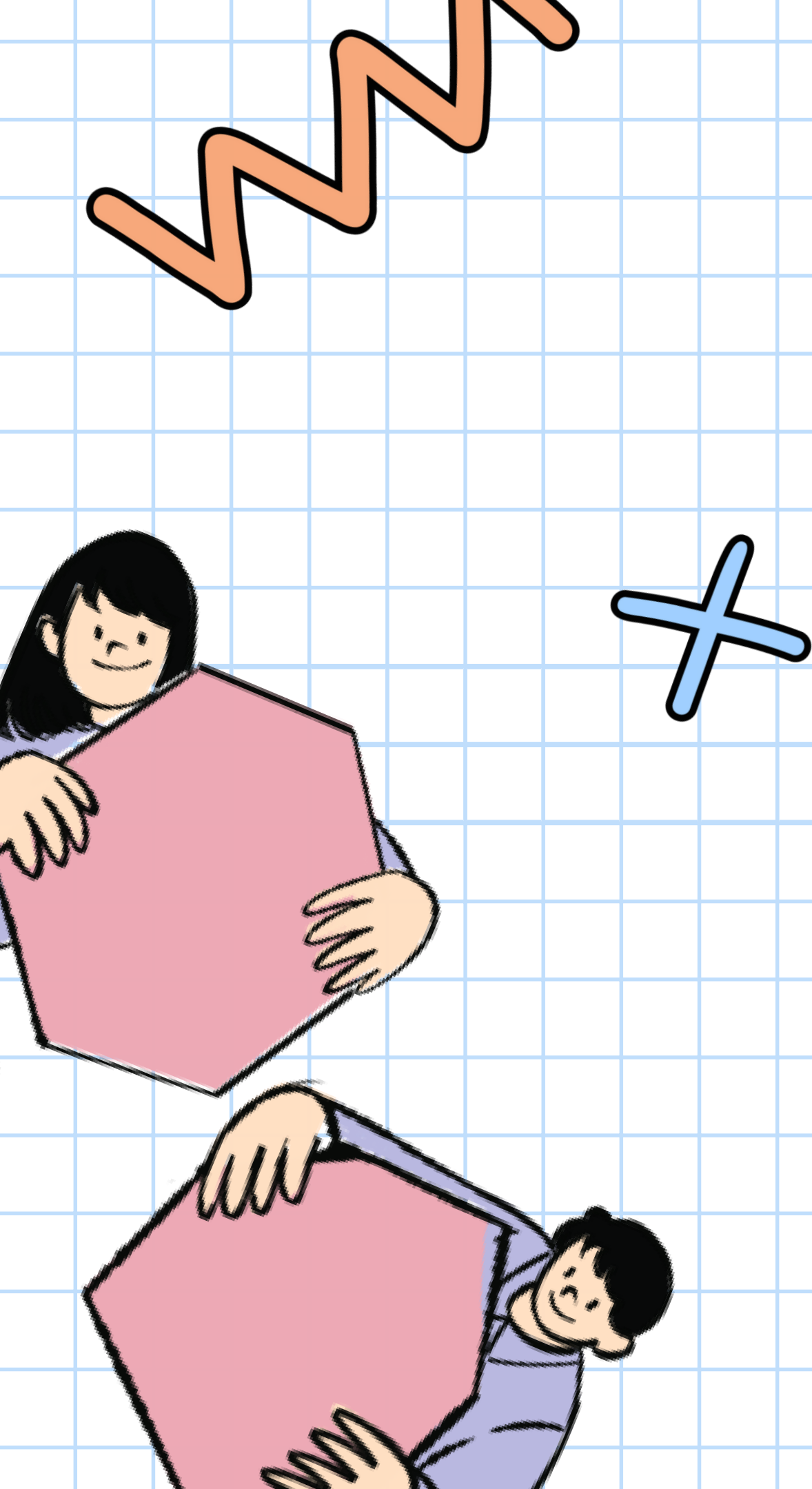


Lenguaje de figuras














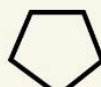





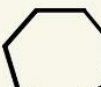


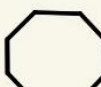







Guillermo, Ingrid y Raquel

Índice

- Maquina virtual basada en pilas
- Ensamblador ejecutado por MV
- Proceso de ejecución



Tokens


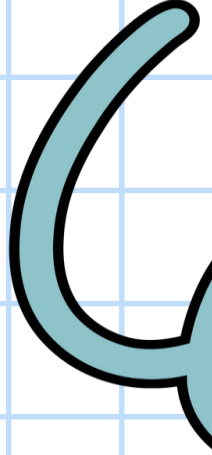
FIGURAS		FIGURAS		FIGURAS	
	+		>		0
	-		<		1
	*		If > Elif		2
	/				3
	%		While		4
	**		For		5
	!=		Print		6
	//				7
	=				8
	()		,		9

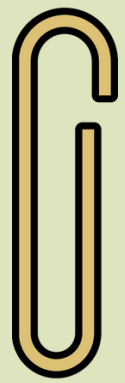
A light blue folder with a darker blue paperclip on the left side. The word ENSAMBLADOR is written in the center in a bold, black, sans-serif font.

ENSAMBLADOR



Cómo funciona la pila en nuestro lenguaje

- En nuestro proyecto usamos una **pila (LIFO)**: los valores se apilan y desapilan por arriba
 - Con la **Notación Polaca Inversa (RPN)**, cada número se mete en la pila y cada operación saca los dos últimos y mete el resultado
 - Esto evita paréntesis y hace que el orden de operaciones se controle automáticamente
- 
- 



Números(solo push)

- Cada número que añadimos se apila
- No hay operaciones todavía, solo entradas

- **Infix(normal)**

- **RPN**

5 2 9 (entradas)

1. Push 5 [5]
2. Push 2 [5, 2]
3. Push 9 [5, 2, 9]

Suma '+'

- Saca los dos últimos valores, los suma y apila el resultado

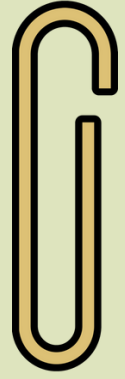
- **Infix(normal)**

5+2

- **RPN**

5 2 +

1. Push 5 [5]
2. Push 2 [5, 2]
3. + \rightarrow 5+2=7; push [7]



Resta '-' (a - b)

- Se sacan los dos últimos y se calcula (a - b)

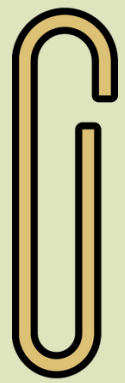
- **Infix(normal)**

8 - 3

- **RPN**

8 3 -

- Push 8 [8]
- Push 3 [8, 3]
- → 8-3=7; push [5]



Multiplicación '*'

- Multiplica los dos valores superiores y mete el resultado

- **Infix(normal)**

2 * 5

- **RPN**

2 5 *

1. Push 2 → [2]

2. Push 5 → [2, 5]

3. → 2 × 5 = 10 → Push [10]



División '/'

- Saca los dos últimos valores, divide el primero entre el segundo y apila el resultado. (En la app, el resultado lo hemos redondeado con `Math.round()`)

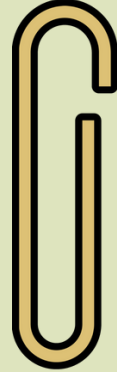
- **Infix(normal)**

7 / 2

- **RPN**

7 2 /

1. Push 7 → [7]
2. Push 2 → [7, 2]
3. / → $7 \div 2 = 3.5$ → redondea → 4 → Push [4]



Módulo '%'

- Devuelve el resto de la división entre los dos valores superiores

- **Infix(normal)**

10 % 3

- **RPN**

10 3 %

1. Push 10 → [10]
2. Push 3 → [10, 3]
3. % → 10 % 3 = 1 → Push [1]



Potencia '**'

- Devuelve el resto de la división entre los dos valores superiores

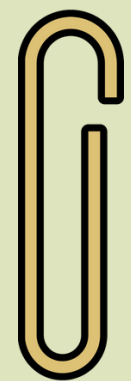
- **Infix(normal)**

2 ** 5

- **RPN**

2 5 **

1. Push 2 → [2]
2. Push 5 → [2, 5]
3. ** → $2^5 = 32$ → Push [32]



Comparaciones ('>', '<', '==', '!=')

- Las comparaciones sacan dos valores y apilan 1 si la condición es verdadera, o 0 si es falsa

- **Infix(normal)**

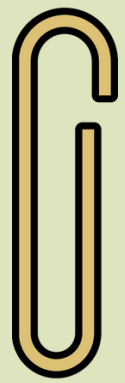
7 > 5

- **RPN**

7 5 >

1. Push 7 → [7]
2. Push 5 → [7, 5]
3. → 7 > 5 → verdadero → Push [1]

(para == sería 1 si son iguales, para != sería 1 si son distintos, etc)



Paréntesis y precedencia

- En RPN no se usan paréntesis
- El orden correcto se logra con la pila y la posición de los operadores

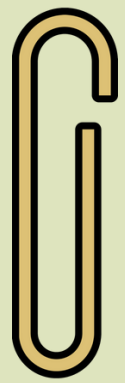
- **Infix(normal)**

$(3 - 1) * 4$

- **RPN**

3 1 - 4 *

1. Push 3 → [3]
2. Push 1 → [3, 1]
3. - → 3-1=2; push → [2]
4. Push 4 → [2, 4]
5. * → 2*4=8; push → [8]



Paréntesis y precedencia

- En RPN no se usan paréntesis
- El orden correcto se logra con la pila y la posición de los operadores

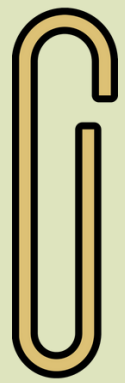
- **Infix(normal)**

$7 / (2 + 5)$

- **RPN**

$7\ 2\ 5\ +\ /$

- Push 7 $\rightarrow [7]$
- Push 2 $\rightarrow [7, 2]$
- Push 5 $\rightarrow [7, 2, 5]$
- + $\rightarrow 2+5=7$; push $\rightarrow [7, 7]$
- / $\rightarrow \text{round}(7/7)=1$; push $\rightarrow [1]$



Paréntesis y precedencia

- En RPN no se usan paréntesis
- El orden correcto se logra con la pila y la posición de los operadores

- **Infix(normal)**

$2 + 5 * 3$ (por precedencia $*$ va antes que $+$)

- **RPN**

$2\ 5\ 3\ *\ +$

1. Push 2 $\rightarrow [2]$
2. Push 5 $\rightarrow [2, 5]$
3. Push 3 $\rightarrow [2, 5, 3]$
4. $*$ $\rightarrow 3*5=15$; push $\rightarrow [2, 15]$
5. $+$ $\rightarrow 2+15=17$; push $\rightarrow [17]$



PRINT

- PRINT muestra el valor superior **sin eliminarlo** de la pila.
- En la app, se ejecuta después de la operación

- **Infix(normal)**

```
2 * 5 ; PRINT
```

- **RPN**

```
2 5 * PRINT
```

1. Push 2 [2]
2. Push 5 [2, 5]
3. * 2*5=10; push [10]
4. PRINT muestra 10 (no pop) [10]

FOR

- FOR n DO ... END repite el bloque n veces
Cada vuelta añade un POP al final para limpiar la pila

- Infix(normal) 8 - 3

- RPN(expandido) 2 5 * PRINT 2 5 * PRINT 2 5 * PRINT (+ POP diferido tras cada vuelta)

1. [V1] Push 2 → [2]
2. [V1] Push 5 → [2, 5]
3. [V1] * → 2*5=10; push → [10]
4. [V1] PRINT → muestra 10 → [10]
5. [V1] POP (fin vuelta) → []
6. [V2] Push 2 → [2]
7. [V2] Push 5 → [2, 5]
8. [V2] * → 2*5=10; push → [10]
9. [V2] PRINT → muestra 10 → [10]
10. [V2] POP (fin vuelta) → []
11. [V3] Push 2 → [2]
12. [V3] Push 5 → [2, 5]
13. [V3] * → 2*5=10; push → [10]
14. [V3] PRINT → muestra 10 → [10]
15. [V3] POP (fin vuelta) → []

El POP por vuelta evita que el
resultado de una iteración
contamine a la siguiente

IF/ELIF/ELSE

- Solo se ejecuta el bloque donde la condición sea verdadera.
Durante la expansión se evalúan las condiciones y se genera solo el bloque elegido

```
IF (3 > 2) DO 7 + 1 ; PRINT END ELSE 9 ; PRINT END
```

- **RPN elegido** `3 2 > 7 1 + PRINT`

1. Push 3 → [3]
2. Push 2 → [3, 2]
3. > → (3 > 2) = 1; push → [1]
4. Push 7 → [1, 7]
5. Push 1 → [1, 7, 1]
6. + → 7+1 = 8; push → [1, 8]
7. PRINT → muestra 8 → [1, 8]

Si la condición fuera 0, se expandiría el bloque ELSE

1) Suma — Círculo negro (ADD)

.entry main

; punto de entrada

main:

PUSH a

; apila el primer operando (a)

PUSH b

; apila el segundo operando (b)

ADD

; (a+b) sustituye a los dos tops

PRINT

; muestra el resultado

HALT

; fin del programa

2) Resta — Rectángulo negro (SUB)

```
.entry main
```

```
main:
```

```
    PUSH a
```

```
    ; minuendo
```

```
    PUSH b
```

```
    ; sustraendo
```

```
    SUB
```

```
    ; realiza a - b
```

```
    PRINT
```

```
    HALT
```

3) Multiplicación — Triángulo negro (MUL)

```
.entry main
```

```
main:
```

```
    PUSH a
```

```
    ; primer factor
```

```
    PUSH b
```

```
    ; segundo factor
```

```
    MUL
```

```
    ; producto a*b
```

```
    PRINT
```

```
    HALT
```

4) División — Cuadrado negro (DIV)

```
.entry main
```

```
main:
```

```
    PUSH a
```

```
    PUSH b
```

```
    DIV
```

```
    PRINT
```

```
    HALT
```

```
; dividendo
```

```
; divisor ( $\neq 0$ )
```

```
; cociente truncado hacia 0
```

5) División entera / Módulo — Heptágono negro (DIV / MOD)

; División entera

.entry main

main:

PUSH a

; dividendo

PUSH b

; divisor

DIV

; cociente entero

PRINT

HALT

; Módulo (resto)

.entry main

main:

PUSH a

; dividendo

PUSH b

; divisor

MOD

; resto a % b

PRINT

HALT

6) Potencia — Pentágono negro (a ** b)

```

.entry main
main:
    POPR R1           ; R1 ← b (exponente)
    POPR R0           ; R0 ← a (base)
    PUSH 1            ; acc ← 1 (acumulador)

pow_loop:
    PUSHR R1          ; push(b)
    PUSH 0            ; push(0)
    CMPL             ; ¿b ≤ 0? → 1 si sí, 0 si no
    JNZ pow_end       ; si (b ≤ 0) salta al final

    PUSHR R0          ; push(a)
    MUL              ; acc = acc * a

    PUSHR R1          ; push(b)
    PUSH 1            ; push(1)
    SUB              ; b - 1
    POPR R1           ; R1 ← b-1
    JMP pow_loop      ; repetir

pow_end:
    PRINT            ; imprime acc
    HALT

```

7) Distinto (!=) — Hexágono negro

.entry main

main:

PUSH a

; primer valor

PUSH b

; segundo valor

CMP

; actualiza banderas con (a ? b)

JZ neq_is0

; si Z=1 (iguales) saltar a 0

PUSH 1

; no son iguales → 1

PRINT

HALT

neq_is0:

PUSH 0

; iguales → 0

PRINT

HALT

HALT

8) Igual (=) — Octágono negro

.entry main

main:

PUSH a

; primer valor

PUSH b

; segundo valor

CMP

; compara

JZ eq_is1

; si Z=1 (iguales) → 1

PUSH 0

; si no, 0

PRINT

HALT

eq_is1:

PUSH 1

; iguales → 1

PRINT

HALT

9) Mayor que (>)

.entry main

main:

PUSH a

PUSH b

SWAP

CMPLT

PRINT

HALT

; candidato a ser mayor

; comparando con b

; intercambio para usar CMPLT

; evalúa $(b < a) \rightarrow 1/0$

10) Menor que (<)

```
.entry main
```

```
main:
```

```
    PUSH a
```

```
    PUSH b
```

```
    CMPLT
```

```
    PRINT
```

```
    HALT
```

; candidato a ser menor

; evalúa $(a < b) \rightarrow 1/0$

11) IF / ELSE — Triángulo / Rombo blancos

```
.entry main
```

```
main:
```

```
    ; deja cond en la cima (1 si verdadero, 0 si falso)
```

```
    PUSH 0
```

```
    ; para normalizar con CMPLE
```

```
    CMPLE
```

```
    ; cond <= 0 ? → 1 si falso
```

```
    JNZ else_lbl
```

```
    ; si falso → salta a else
```

```
    ; --- bloque A (then) ---
```

```
    ; ... instrucciones ...
```

```
    JMP end_if
```

```
    ; saltar el else
```

```
else_lbl:
```

```
    ; --- bloque B (else) ---
```

```
    ; ... instrucciones ...
```

```
end_if:
```

```
    HALT
```

12) WHILE — Pentágono blanco

```
.entry main
main:
while_top:
    ; deja cond (1/0) en la cima para evaluar
    PUSH 0
    CMPL                     ; cond <= 0 ?
    JNZ while_end            ; si falso → salir del bucle

    ; --- cuerpo del bucle ---
    ; ... instrucciones ...
    JMP while_top            ; repetir

while_end:
    HALT
```

NO INCLUIDO EN LA MÁQUINA VIRTUAL POR CARENCIA DE ASIGNACIÓN DE VARIABLES

13) FOR — Hexágono blanco (con registros)

```
.entry main
main:
    POPR R1                ; R1 ← N (límite superior)
    PUSH 0
    POPR R0                ; R0 ← 0 (contador i)

for_top:
    PUSHR R0
    PUSHR R1
    CMPLT                  ; ¿i < N? → 1/0
    PUSH 0
    CMPL                    ; normaliza a 1/0 para JNZ
    JNZ for_end            ; si 0 (falso) → salir

; --- cuerpo: usar i con PUSHR R0 si se necesita ---
; ... instrucciones ...

PUSHR R0
PUSH 1
ADD
POPR R0                    ; i++
JMP for_top                ; siguiente iteración

for_end:
    HALT
```




14) PRINT — Octágono blanco

.entry main

main:

PUSH x

PRINT

HALT

; valor a mostrar

; imprime el tope