

PuréScript

Documentación de Lenguaje de Scripting de Tubérculos



Papita con Puré · v1.1

Ilustración por amamitsu kousuke

Guía

Este capítulo cubre temas importantes tanto para gente que nunca ha programado como gente que quiere saber rápidamente cómo programar en PuréScript.

Si ya sabes programar en PuréScript, puedes revisar el [capítulo de Referencia](#) o ejercitarte creando programas en el [capítulo de Práctica](#). Si no conoces algún concepto de informática, puedes revisar el [capítulo de Glosario](#).

Índice de Capítulo

<i>Guía</i>	1
Introducción	3
Objetivos	3
Hola Mundo	4
Tubérculos y PuréScript	4
Gramática	5
Sentencias	5
Análisis Léxico e Interpretación	6
Comentarios	6
Declaraciones	7
Variables	7
Tipos	8
Bloques y Ámbito de Variables	10
Declaraciones y Asignaciones	10
Conflictos y Similitudes	12
Ámbito de Variables	13
Sentencias de asignación adicionales	13
Componentes Fundamentales	14
Expresiones	14
Palabras Reservadas	15
Operadores	15
Tipos de Operador	18
Expresiones de Conversión	20
Literales	21
Expresiones de Acceso a Miembro	25
Bloques y Estructuras de Control	29
Profundizando el Concepto de Bloques	29
Ocultación de Variables	31
Estructuras de Control	32
Estructuras Condicionales	33

Operadores Lógicos	35
Estructuras Iterativas	41
Otras Sentencias de Control	46
Funciones	47
Expresión de Llamado	47
Argumentos de Función	48
Sentencia EJECUTAR	49
Clasificación de Funciones	49
Expresión de Función	50
Retorno de Valores	51
Paso de Parámetros por Valor y por Referencia	53
Recursividad	55
Funciones Anónimas	57
Funciones como Parámetros	57
Ejemplo para Más Tarde	59
Ámbito de Función	60
Funciones Anidadas	61
Funciones como Retorno	61
Múltiples Funciones Anidadas	62
Mismo Identificador en Diferentes Funciones	63
Funciones Nativas	64
Métodos	65
Funciones Lambda	71
Expresiones de Secuencia	71
Métodos Nativos de Orden Superior	72
Recibiendo Entradas de Usuario	76
Primera Ejecución y Ejecuciones Subsecuentes	77
Sentencia LEER	77
Entradas de Usuario Opcionales	81
Limitando Entradas de Usuario	82
Entradas Extensivas	83
Formatos de Entrada	85
Trabajando con Marcos	87
Métodos de Marco	88
Limitaciones de Marcos	89
Persistencia de Datos	90
Guardado de Datos	90
Carga de Datos y Declaración Condicional	90
Borrando Datos Guardados	93
Limitaciones de Datos Guardados	94

Introducción

PuréScript (**PS**) es un lenguaje de programación procedural, interpretado, dinámico y débilmente tipado para **Bot de Puré**. Se usa para crear comandos personalizados de servidor (conocidos como “Tubérculos”) y permite computar órdenes que manipulan su comportamiento y la respuesta ofrecida por los mismos.

Un programa PuréScript válido debe enviar al menos un mensaje o dato.

Objetivos

¿No sabes programar? ¿Sabes programar pero no en PuréScript? No te preocupes, esta guía arranca desde el nivel más básico para ser accesible tanto a principiantes como a gente que busca un empujón rápido, y luego transiciona suavemente a conceptos más complejos. El objetivo de la misma es informarte sobre los casos de uso más frecuentes del lenguaje y darte una noción general de sus capacidades.

Si ya tienes experiencia en programación, puede que igual quieras leer con detenimiento las primeras páginas. Dicho esto, hay varios conceptos que puedes intuir por tu cuenta luego de superar la barrera de la nueva sintaxis.

Después de leer esta guía con atención, entenderás este código y más:

```
CARGAR factorialDe con Función(núm)
    SI núm excede 1
        DEVOLVER núm * factorialDe(núm - 1)
    SINO
        DEVOLVER núm
    FIN
FIN

REPETIR 3 veces
    //Ejemplo: 6! = 6 * 5 * 4 * 3 * 2 * 1 = 720
    LEER Número fac con 6
    ENVIAR factorialDe(fac)
FIN
```

Hola Mundo

El siguiente código hace que se envíe el mensaje “*Hola mundo!*” en el chat:

```
ENVIAR "¡Hola mundo!"
```

Tubérculos y PuréScript

Para usar PuréScript, necesitas crear un **Tubérculo avanzado**. Hazlo con el comando **p!tubérculo**, así →

```
p!tubérculo --crear --script ...
ENVIAR "Hola mundo"
...
```

`--crear` indica que quieres crear un nuevo **Tubérculo**, mientras que `--script` indica que quieres que sea **avanzado**. Si no puedes escribir `\`, puedes teclear **Alt + 96**.

También puedes abbreviar `--crear` y `--script` como `-cs`, y **p!tubérculo** como **p!t**, así →

```
p!t -cs ...
ENVIAR "Hola mundo"
...
```

Por último: el código no se ve tan bien con ese color, así que coloquemos `arm` luego de los primeros `\``` (sin espacios) así →

```
p!t -cs ```arm
ENVIAR "Hola mundo"
...
```

Pequeña intermisión...

Para saltar un renglón en Discord sin enviar el mensaje, puedes presionar **Shift+Enter**.

Sin más preámbulos, el resto de la guía se concentrará en cómo escribir código, conceptos de programación útiles para el lenguaje, consejos y recomendaciones.

Gramática

PuréScript tiene una sintaxis que asemeja al español y toma elementos de lenguajes como [C](#). La sintaxis está pensada para ser fácil de leer y escribir para principiantes.

PuréScript **no distingue mayúsculas, minúsculas ni tildes**, excepto:

- En [identificadores](#) (o nombres) de variables y funciones, que consideran símbolos exactos para distinguir unos de otros.
- En [Texto](#), cuyo valor son los símbolos exactos utilizados.

Sentencias

Un programa no es mucho más que un “paso a paso” de cosas por hacer. En ese sentido, es muy similar a una lista de tareas. Cada paso es una instrucción para hacer algo específico.

En PuréScript, las instrucciones se llaman **sentencias**. Cada sentencia comienza con un **indicador de sentencia**, que determina cómo será interpretada por el lenguaje.

Cosas por hacer:

1. **Sacar** pan
2. **Sacar** cuchillo
3. **Sacar** queso untable
4. **Dividir** pan con cuchillo
5. **Sacar** queso con cuchillo
6. **Deslizar** cuchillo en pan
7. **Entregar** pan con queso

Nota: Estas sentencias no son reales

Los indicadores de sentencia suelen ser verbos, como [CARGAR](#), [SUMAR](#) (veremos estas luego), o el ya visto [ENVIAR](#). Estos definen la estructura y comportamiento de las sentencias.

En este documento, los *indicadores de sentencia* se escriben en mayúsculas y se colorean de **naranja fuerte**.

Análisis Léxico e Interpretación

PuréScript lee texto de izquierda a derecha y lo convierte en una serie de **Tokens**.

Cualquier **espacio**, **salto de renglón** o **sangría** generalmente se ignora, a menos que *separen dos palabras* o estén dentro de un **Texto**. Por ejemplo:

- Dos palabras — CARGAR Texto **NO ES** lo mismo que CARGAR Texto.
- Texto — "Tengo 4 perros" **NO ES** lo mismo que "Tengo4perros".
- Símbolo — miembro → propiedad **ES** lo mismo que miembro → propiedad.

Los **indicadores de sentencia** como tal son Tokens; y cosas como números, textos, identificadores, operadores, paréntesis, comas, etc. también son Tokens.

Comentarios

Los **comentarios** son texto que puedes agregar a tu código para que lean otras personas. No afectan el proceso. Existen 2 formas de realizar comentarios sobre tu código.

Puedes usar barras dobles (//) para hacer un **comentario de línea**.

```
//Esto es un comentario, y no influye en el código  
//El siguiente código envía "¡Hola mundo!" en el chat  
ENVIAR "¡Hola mundo!"
```

La sentencia **COMENTAR** permite realizar un comentario como un Texto entre comillas dobles. Si no se usan comillas dobles, funciona idéntico a un comentario de línea.

```
COMENTAR "Esto es un comentario,  
y no influye en el código"  
  
COMENTAR "El siguiente código envía '¡Hola mundo!' en el chat"  
ENVIAR "¡Hola mundo!"  
  
COMENTAR Este es un comentario de Línea, idéntico a "//"
```

Declaraciones

Vas a estar trabajando con **valores** todo el rato. Estos son *cualquier dato* que puedes usar en tu programa.



547

puntos
VARIABLE

Por sí solos, los valores sueltos no son muy útiles, pero puedes guardarlos en **variables**. Una variable es básicamente un **nombre** que contiene un valor, dicho valor puede *cambiarse o variar...* ¡por eso es variable!

PuréScript ofrece 2 sentencias principales para **declarar** y **asignar** variables:

CREAR

Declara una nueva variable con un **valor por defecto** (véase [Tipos](#) abajo).

CARGAR

Declara una variable y le asigna un valor expresado. Si la variable ya estaba declarada, solamente le asigna el valor.

Variables

Puedes usar variables como nombres simbólicos para representar valores en tu código. Estos *nombres de variable*, llamados [identificadores](#), siguen ciertas reglas:

- Siempre comienzan con una letra o guion bajo (`_`).
- El resto de caracteres además pueden ser números.
- **No** pueden contener espacios (el espacio delimita el nombre).
- **No** pueden llamarse igual que una palabra reservada ([indicador de sentencia](#), operadores, [tipos de variable](#), etc).
- **Distinguen** mayúsculas, minúsculas y tildes (como se mencionó antes).

Todas las declaraciones de símbolos propios en PuréScript son de *variables*. **No se pueden declarar constantes** (un nombre cuyo valor no se puede cambiar).

Los *identificadores de variable* son celeste atenuado en el código de este documento.

Tipos

↑ PuréScript cuenta con **6 tipos de valor**. 3 de los cuales representan *datos básicos*, y otros 3 que representan *estructuras complejas* que pueden estar asociadas a múltiples datos. Existe el valor especial `Nada` para representar, básicamente, “ningún valor”.

Existe un **indicador de tipo** para cada uno de los tipos que se listan a continuación, escritos igual al nombre del tipo. Estos cumplen múltiples propósitos a lo largo del lenguaje, y se colorean de verde en el código de este documento.

Número

Tipo [primitivo](#). Representa un número real.

Valor por defecto: `0`.

42

Texto

Tipo [primitivo](#). Representa cualquier tipo de texto, entre comillas dobles.

Valor por defecto: `""` (un texto vacío).

“Aa”

Lógico

Tipo [primitivo](#). Representa un valor lógico: `Verdadero` o `Falso`.

Valor por defecto: `Falso`.

¿?

Lista

Tipo de [estructura](#). Representa un conjunto *ordenado* de elementos, enumerados/indizados desde `0`.

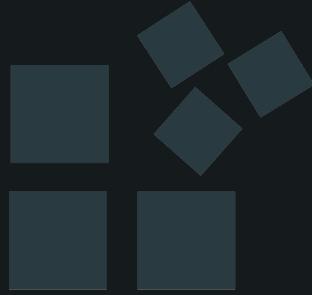
Valor por defecto: Lista vacía, o sin elementos (ningún índice para un elemento).



Registro

Tipo de [estructura](#). Representa un conjunto de *pares relacionados* (o entradas).

Cada entrada consta de una clave y un valor. La clave es **única** en el conjunto y se usa para *acceder* su respectivo valor.



Puedes pensar en ello como un diccionario: las claves serían palabras y los valores serían sus respectivas definiciones.

Valor por defecto: Registro vacío, o sin entradas (ninguna clave para un valor).

Marco

[Estructura](#) especial para PuréScript. Representa la estructura de un **marco embebido de mensaje** (Embed) de [Discord](#).



Se utiliza **sólo para enviar un marco**, y sus propiedades solo pueden asignarse por medio de *métodos de Marco* (lo veremos luego).

Valor por defecto: Marco vacío (envío inválido) — sin título, descripción, autor, pie, fecha ni campos; color por defecto.

Más tarde veremos que, en realidad, existe un tipo de valor adicional. Sin embargo, no estamos listos para comprender eso todavía.

Bloques y Ámbito de Variables

Los **bloques** son conjuntos de sentencias. *Pueden ejecutarse*, lo cual ejecuta en orden las sentencias que contienen.

Al ejecutar un **Tubérculo**, el primer bloque que se ejecuta siempre es el **bloque Programa**, que contiene **todas** las sentencias del código que escribimos.

El bloque Programa en particular es *implícito* y está presente en todos los programas de PuréScript, a diferencia de los **bloques declarados explícitamente** que veremos [luego](#).

Los bloques *también son sentencias*, así que pueden haber **bloques dentro de bloques**.

Todos los bloques ofrecen lo que se conoce como un **ámbito de variables**. Todas las **variables** viven en un **ámbito**. Además, los ámbitos, al igual que los bloques, pueden **vivir dentro de otros ámbitos** y *usarlos*. Profundizaremos sobre este concepto más adelante.

Declaraciones y Asignaciones

La sentencia `CREAR` declara una variable en el ámbito actual, con el *valor por defecto correspondiente al tipo indicado*:

```
//Crea una variable "dinero", con el valor por defecto: 0
CREAR Número dinero

//Crea la variable "nombre", con un texto vacío/sin caracteres
CREAR Texto nombre

//Crea la Lista "frases", como una lista vacía
CREAR Lista frases
```

Si la variable ya estaba declarada en este **ámbito**, se alza un error.

Bloque Programa

Bloque

Bloque

Ámbito Local

Ámbito Local

Bloque

Ámbito Local

Ámbito Global

La sentencia CARGAR **solo declara variables que no estén declaradas**. Además, a diferencia de CREAR, que asigna un *valor por defecto*, esta les da el *valor que indiques*:

```
//Declara La variable "contador" de tipo Número y le asigna 3
CARGAR contador con 3

//Crea La Lista "frases" y le asigna 3 valores de Texto
CREAR Lista frases
CARGAR frases con Lista "Hola", "Qué tal", "Adiós"
```

Sintaxis de asignación

La palabra reservada **con** es usada en una multitud de sentencias. Generalmente indica que, de alguna forma, se debe asignar el **valor de la expresión** de la *derecha* al **identificador** de la *izquierda*.

```
CARGAR <identificador> con <expresión>
```

El identificador de la izquierda se conoce como “receptor” – porque recibe un valor – y la expresión de la derecha se conoce como “recepción” – porque es el valor que se recibe.

Volviendo al ejemplo de más arriba: CARGAR contador con 3, podemos leerlo en español como “*toma una variable llamada ‘contador’ y cárgale el valor 3*”.

Las variables deben ser declaradas **antes** de ser utilizadas (leído de arriba a abajo):

```
//;Cuidado! "unValor" va a valer Nada, porque "otroValor" todavía no ha
//sido declarada en el punto que se la menciona
CARGAR unValor con otroValor
CARGAR otroValor con 3
```

También puedes usar expresiones matemáticas y demás como recepción:

```
//Asigna el valor 7 a "unValor", porque 4 + 3 = 7
CARGAR unValor con 4 + 3
```

Conflictos y Similitudes

Usar `CREAR` y `CARGAR` para declarar y darle valor a una variable es lo mismo que simplemente usar `CARGAR`, **solo si la misma no estaba ya declarada**.

CARGAR **solo** declara una variable *si la misma no ha sido declarada*, mientras que CREAR **siempre** va a intentar declararla. Si se intenta **declarar el mismo nombre** en el **mismo lugar** más de una vez, ocurrirá un error.

```
//No se alza un error, este CARGAR solo le asigna al "unTexto" ya creado  
CREAR Texto unTexto  
CARGAR unTexto con "truz"
```

```
//No se alza un error. El primer CARGAR declara y el segundo no
CARGAR unTexto con "Había una vez..."
CARGAR unTexto con "truz"
```

Puede que encuentres que esto se llama “declaración automática” en la Referencia.

A tener en cuenta...

A menos que tengas un motivo claro para usar CREAR (como veremos más adelante), suele ser preferible usar CARGAR para declarar variables.

Ámbito de Variables

Las variables siempre se declaran en un cierto **ámbito**. El ámbito de una variable define hasta cuándo vive y qué partes del código pueden accederla.

Una variable puede pertenecer a uno de los siguientes ámbitos al ser declarada:

- **Ámbito global** — La variable está declarada directamente sobre el **bloque Programa**, o sea que es accesible por el resto del **Tubérculo**.
- **Ámbito de bloque** — La variable es accesible dentro del bloque actual.
- **Ámbito de sentencia** — La variable es accesible en la sentencia actual.
- **Ámbito de Función** — Similar al de bloque. Lo veremos después.

Una variable declarada en un ámbito será **destruída** cuando el mismo finalice. Declarar una variable directamente sobre el **bloque Programa** hace que la misma viva *desde que se declara hasta que el programa termina de ejecutarse*, mientras que una variable declarada en un bloque interno vivirá desde su declaración hasta que el bloque finalice.

```
//Esta variable permanecerá viva por el resto del Programa  
CARGAR unValor con 3
```

Cuando una variable se destruye, es **eliminada y olvidada** por completo. Lo cuál significa que referirse a su identificador en ciertas ocasiones **alzará un error** porque la variable ya no existe. Esto también significa que puedes declarar *otra* variable con el mismo identificador luego de que la anterior haya terminado su vida.

Sentencias de asignación adicionales

Sentencia	Fórmula	Efecto
CARGAR x con e	x = e	Auto-declara. Asigna el valor de e a la variable x.
SUMAR x con e	x = x + e	Calcula el valor de x más e y lo asigna a x.
RESTAR x con e	x = x - e	Calcula el valor de x menos e y lo asigna a x.
MULTIPLICAR x con e	x = x * e	Calcula el valor de x por e y lo asigna a x.
DIVIDIR x con e	x = x / e	Calcula el valor de x sobre e y asigna el resultado a x.

Componentes Fundamentales

PuréScript es, a fin de cuentas, un conjunto de instrucciones (sentencias) que se ejecutan de arriba hacia abajo para cumplir alguna tarea. Las **sentencias**, junto a las **expresiones** y **palabras reservadas**, son **todo** lo que compone al lenguaje a un nivel básico.

La vista general y rápida es:

- Un **programa** de PuréScript contiene **sentencias**.
- Una **sentencia** puede contener **expresiones** y **palabras reservadas**
- Una **expresión** puede ser una **operación**.
- Una **operación** contiene **operandos** y operadores.
- Los **operandos** son expresiones.

¡Tranquilo! Debajo lo vemos más despacio...

Expresiones

Una **expresión** es cualquier pedacito de código del cual se obtiene un *valor de resultado*. Dicho formalmente: las expresiones pueden ser *evaluadas*.

Las expresiones son una parte esencial del lenguaje. Casi todas las sentencias hacen uso de expresiones para especificar cómo realizar una acción. Una expresión **no tiene efectos secundarios por sí sola**. A modo de ejemplo, algunos usos de expresiones son:

- **CREAR** declara una variable cuyo nombre es dado por una *expresión de identificador*.
- **CARGAR** asigna la *expresión* derecha a la variable indicada en la *expresión de identificador* de la izquierda.
- **ENVIAR** envía un mensaje de Discord en base a la *expresión* dada.

```
CREAR <identificador>
CARGAR <identificador> con <expresión>
ENVIAR <expresión>
```

Palabras Reservadas

Son aquellas palabras que, de no estar reservadas, podrían ser un identificador.

Pueden no representar nada *por sí mismas*, y aquellas que no lo hacen se las usa en conjunto con ciertas sentencias.

Dado que las palabras reservadas **no** son identificadores, **no distinguen mayúsculas de minúsculas ni tildes**.

Todos los indicadores de sentencia son palabras reservadas, pero no todas las palabras reservadas son indicadores de sentencia. La misma relación aplica para los indicadores de tipo. Además, algunas palabras reservadas son operadores y vice-versa.

Excluyendo lo del párrafo anterior, el resto de palabras reservadas son:

- `verdadero`
- `nada`
- `en`
- `opcional`
- `falso`
- `con`
- `desde`
- `veces`

Por ejemplo: hemos visto que la palabra clave `con` puede usarse para separar el receptor y la recepción en las sentencias de asignación.

Las palabras reservadas de la anterior tabla, con la excepción de `Verdadero`, `Falso` y `Nada` se colorearán de gris claro para el código de este documento.

Operadores

Un *operador* define qué *operación* debe realizarse sobre una o más expresiones, referidas en este caso como *operandos*. Las operaciones utilizan los operandos que las conforman para dar un valor de resultado.

¡Las operaciones son expresiones! O sea que **pueden ser operandos** de otra *operación*.

Los operadores se colorean de gris claro en el código de este documento.

Precedencia y asociatividad de operadores

Cuando se encuentran múltiples operaciones juntas, el **orden en el que se evalúan** es determinado por la precedencia de sus operadores. La precedencia de operadores define qué operaciones tienen prioridad sobre otras (literalmente “preceden” a otras) a la hora de evaluarlas. **Los operadores de mayor precedencia se vuelven operandos de los operadores de menor precedencia.**

Por ejemplo, en matemática existe la regla de *P.E.M.D.S.R.*, que define entre otras cosas que la multiplicación se realiza antes que la suma (tiene *mayor precedencia* que esta):

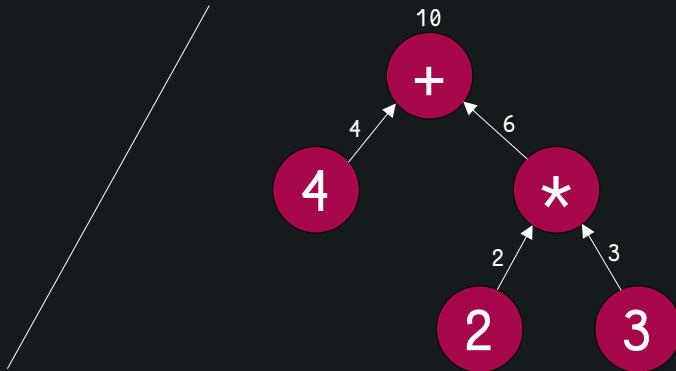
```
CARGAR a con 1 + 2 * 3 //a = 1 + (2 * 3) = 1 + 6 = 7  
CARGAR b con 2 * 3 + 1 //b = (2 * 3) + 1 = 6 + 1 = 7
```

Cuando dos operaciones son de la misma precedencia, se evalúan «*de izquierda a derecha*» o «*de derecha a izquierda*» según su *asociatividad*. Por ejemplo: la suma y resta tienen la **misma precedencia**, así que se usa su asociatividad a la izquierda:

```
ENVIAR 1 + 2 - 3 + 4 //((1 + 2) - 3) + 4 = (3 - 3) + 4 = 0 + 4 = 4  
ENVIAR 1 - 2 + 3 - 4 //((1 - 2) + 3) - 4 = (-1 + 3) - 4 = 2 - 4 = -2
```

$$4 + 2 * 3$$

(= 10)



Esto aplica a **cualquier tipo de operador**, no solo a operadores *matemáticos*.

Próximamente se verán diferentes tipos de operadores. También puedes encontrar un listado completo y detallado de todos los operadores en la [Referencia](#).

Clasificación de operadores

Un **operador** puede ser *prefijo*, *sufijo* o *infijo*:

- **Prefijo** — se coloca antes de los operandos.

`operador operando1`

- **Sufijo** — se coloca después de los operandos.

`operando1 operador`

- **Infijo** — se coloca entre los operandos. En operaciones binarias.

`operando1 operador operando2`

Así mismo, un operador puede ser *unario*, *binario* o *ternario* (este *último* no está en PS):

- **Unario** — la operación acepta un solo valor. Este tipo de operación solo se puede representar con operadores prefijos o sufijos.

`operador operando1`

`operando1 operador`

- **Binario** — la operación acepta dos valores. Generalmente infijo.

`operando1 operador operando2`

Por ejemplo, en una operación de suma, el operador es “+”. Este operador es binario infijo, o sea que toma dos operandos numéricos a su alrededor. Ante la expresión “2 + 3”, sabemos que hay que realizar una suma entre los operandos. Al calcular, evaluamos a 5.

PuréScript principalmente tiene *operadores unarios prefijos* y *operadores binarios infijos*:

`operador argumento`
`operando1 operador operando2`

A partir de ahora, se les puede llegar a referir de forma corta como operadores prefijos y binarios respectivamente.

Tipos de Operador

En el capítulo de [Bloques y Estructuras de Control](#) investigaremos más operadores como los *relacionales* y *conectores*. De momento, veremos los siguientes:

- [Operadores de agrupación](#)
- [Operador de concatenación](#)
- [Operadores aritméticos](#)

Operadores de agrupación

Al igual que en matemática, puedes usar paréntesis para cambiar el orden de evaluación de las operaciones. Los [operadores de agrupación](#) (()) tienen la precedencia más alta y la operación de agrupación evalúa al resultado de la expresión entre los paréntesis.

```
CARGAR a con 1 + 2 * 3 //a = 1 + (2 * 3) = 1 + 6 = 7
CARGAR b con (1 + 2) * 3 //b = (1 + 2) * 3 = 3 * 3 = 9
```

Operador de concatenación

Cuando **al menos uno** de los operandos es un **Texto**, el [operador binario +](#) *concatena ambas expresiones en un solo Texto* – la primera antes de la segunda. Cualquier expresión que no resuelva a un Texto será **convertida** a su *representación de Texto*.

```
///"Puntos: 100"
ENVIAR "Puntos: " + 100

///"Tienes 100 puntos"
ENVIAR "Tienes " + 100 + " puntos"

ENVIAR "2" + 2      //"22"
ENVIAR Texto 2 + 2 //"4"

///"El dicho es Verdadero"
ENVIAR "El dicho es " + Verdadero

///"Esto es (RARO)"
ENVIAR "Esto es " + (Lista "R", "A", "R", "O")
```

Operadores aritméticos

Si **ninguno** de los dos operandos es un **Texto**, el operador binario + suma ambas expresiones de forma aritmética. Cualquier expresión que no resuelva a un Número será **convertida** a su *representación de Número*.

```
CARGAR var con 3 + Falso //var = 3  
CARGAR var con 1000 + Verdadero //var = 1001
```

Los operadores aritméticos toman operandos numéricos y evalúan a un resultado numérico. Además de la operación de suma (+) mencionada arriba, las operaciones aritméticas más comunes son la resta (-), la multiplicación (*) y la división (/).

A diferencia del operador +, el resto de operadores aritméticos **siempre** usan la representación numérica de ambos operandos, incluso si son Textos.

Puedes leer más sobre operadores aritméticos en la [Referencia](#).

```
ENVIAR 3 - 1 //Envía "2"  
ENVIAR 6 / 2 //Envía "3"  
ENVIAR 5 * 0.5 //Envía "2.5"
```

También se cuenta con los siguientes operadores aritméticos:

Operador	Descripción	Ejemplos
%	Operador binario. Devuelve el resto de una división. (Esta operación se conoce como “módulo”).	19 % 4 = 3 32 % 4 = 0
+ (unario)	Operador unario. Intenta convertir el operando a un número (si todavía no lo es).	+ "2" = 2 + Verdadero = 1
- (unario)	Operador unario. Devuelve la negación del operando.	-4
^	Operador binario. Si tenemos una expresión de x^y , calcula la potencia de x a la y .	2 ^ 3 = 8 2 ^ 4 = 16

Expresiones de Conversión

Previamente vimos que algunos operadores **convierten** sus operandos a un tipo específico antes de realizar una operación. Veamos ahora una forma de convertir explícitamente.

La expresión de conversión consiste de un *indicador de tipo primitivo* (Número, Texto, Lógico) y la **expresión** cuyo valor se desea convertir. ¡Así de simple!

```
CARGAR a con 42          //c = 42
CARGAR b con Texto 42    //b = "42"
CARGAR c con Lógico 42  //c = Verdadero

CARGAR d con "Un texto" //d = "Un texto"
CARGAR e con Número "Un texto" //e = 0
CARGAR f con Número "30"    //f = 30
CARGAR g con Lógico "Un texto" //g = Verdadero

CARGAR h con Verdadero   //h = Verdadero
CARGAR i con Número Verdadero //i = 1
CARGAR j con Número Falso  //j = 0
CARGAR k con Texto Verdadero //k = "Verdadero"

ENVIAR (Texto 2) + 2 //Envía "22"
ENVIAR Texto (2 + 2) //Envía "4"
```

Estos son solo algunos ejemplos de conversión, puedes ver todas las conversiones primitivas posibles en las secciones de *tipo de la Referencia*:

- [Conversiones de Número](#)
- [Conversiones de Texto](#)
- [Conversiones de Lógico](#)

Literales

Los literales son **expresiones** que representan valores, pero no son variables. Son valores explícitos que escribes *literalmente* en tu código. PuréScript cuenta con los siguientes literales, que afrontaremos en esta sección:

- [Literal de Número](#)
- [Literal de Texto](#)
- [Literal de Lógico](#)
- [Literal de Lista](#)
- [Literal de Registro](#)
- [Literal de Nada](#)

Los Marcos, si bien vistos como un *derivado* de los Registros, están **prohibidos** de ser representados de forma literal.

Las imágenes y otros archivos solo pueden representarse por medio de **Textos**, con enlaces a recursos web de acceso público.

Literal de Número

↑ Los literales de [Número](#) en PuréScript incluyen enteros y decimales sin signos.

Todos los Números en PuréScript se interpretan en [base-10](#), significando que no hay expresiones numéricas [binarias](#) ni [octales](#) ni [hexadecimales](#). Solo [decimales](#).

En el caso de querer Números negativos, se usa el prefijo aritmético `-` por delante del Número. Por ejemplo, la expresión `-43.72` se interpreta como un operador unario `-` aplicado al operando numérico: `43.72`.

Tanto el prefijo `-` como el prefijo `+` pueden usarse para tratar un valor como un Número (o sea, convertido a su *representación numérica*).

Los literales de Número se colorean de **rojo** en el código de este documento.

Los **enteros** solo contienen dígitos:

0, 42, 5, 759, 02062003, 1674663241

Los **decimales** contienen dígitos y un punto decimal:

0.5, 42.3, 9.99999999, 759.01, .504

Literal de Texto

[Volver a Guía - Gramática – Sentencias](#)

↑ Un literal de **Texto** se expresa con 0 ó más caracteres encerrados entre comillas dobles. Se colorean con verde lima claro en este documento.

```
"Esto es texto"  
"Bastante textual de tu parte"  
"Quiero café"  
"348957432895734"  
""
```

Secuencias de escape en Textos

Las combinaciones de barra inversa (\) + cualquier carácter son "secuencias de escape". Se las considera como un solo carácter y señalan una “acción” dentro del Texto.

Tanto la barra inversa como el carácter subsecuente son descartados del Texto resultante, y en su lugar se inserta alguna de las siguientes acciones:

Carácter	Acción
\n	“Nueva línea”. Inserta un salto de renglón en el lugar.
\t	“Tabulación”. Inserta una sangría en el lugar.
\"	Inserta el carácter de comillas dobles en el lugar, sin delimitar el Texto.
\\\	Inserta el carácter literal de barra inversa en el Texto.

Por ejemplo:

```
"¿Ya te conté de la vez que conocí a \"Papita con Puré\"?"  
"¡Esto es un renglón!\n¡Esto es otro renglón!!"  
"¿Te soy sincero? \\ Separar con estas barras puede quedar bien."
```

Intentar escapar un carácter que no esté en la tabla de arriba alzará un error.

Literal de Lógico

↑ Los valores de tipo **Lógico** representan un valor de verdad lógica. Esto solo puede ser uno de dos estados: Verdadero o Falso. Son color rojo en el código de este documento.

```
Verdadero  
Falso
```

Si sabes de programación o álgebra de Boole, probablemente reconozcas que este tipo de valor se suele llamar “**booleano**”.

Literal de Lista

↑ Una expresión literal de **Lista** consiste del indicador de tipo **Lista**, seguido de 0 ó más expresiones separadas por comas – cada una representa un **elemento** de **Lista**.

Un valor de **Lista** creado a partir de un literal de **Lista** contendrá los valores de todas las expresiones como *elementos de la misma*, ordenados según se expresaron, y tendrá una clave **largo** cuyo valor será la cantidad de valores.

```
CARGAR juegos con Lista "Ajedrez", "Damas", "Truco"
```

En el anterior ejemplo, se crea una **Lista** **juegos** con 3 elementos (**largo = 3**). Las **Listas** se indexan desde 0, así que el índice 0 contiene “Ajedrez”, el índice 1 contiene “Damas” y el índice 2 contiene “Truco”.

Veremos cómo acceder índices y claves más adelante.

Puedes señalizar espacios vacíos en la Lista con [comas extra adyacentes](#). También puedes agregar una [coma al final](#) si así lo deseas, sin modificar sus elementos:

```
CARGAR números con Lista 1, 2, 3, , , 4, 5, 6,  
ENVIAR Texto números //Envía "(123NadaNada456)"
```

A partir de las 2 comas extra al final, se empiezan a agregar espacios vacíos al final.

Literal de Registro

↑ Un literal de [Registro](#) es una sucesión de 0 ó más pares de identificadores y expresiones, llamados claves y valores, unidos por el [operador de dos puntos :](#), cada uno de los cuales representa un *miembro* o una **entrada** de Registro. Todos estos pares de clave-valor son precedidos por el [indicador de tipo Registro](#), y se separan por comas.

```
CARGAR juego con Registro  
nombre: "Terraria",  
género: "Acción/Aventura",  
plataformas: (Lista "PC", "Consolas", "Móvil"),  
salida: 2011,
```

Puedes poner una [coma al final](#) que no se toma en cuenta, pero no puedes colocar espacios vacíos con comas extra como harías con las Listas, ya que los Registros en realidad no se ordenan de una forma determinada.

Literal de Nada

↑ Un literal de [Nada](#) representa y evalúa al valor especial [Nada](#).

[Nada](#) es su propio tipo de valor y al mismo tiempo es un valor sin tipo. Aparece en múltiples ocasiones para representar “ningún valor”. Por ejemplo, cada vez que se busca un identificador inexistente, este se evalúa como [Nada](#).

Se escribe igual que se llama, y es color rojo en el código de este documento:

```
Nada
```

Expresiones de Acceso a Miembro

Como ya vimos, tanto los valores de **Listas** como los de **Registros** pueden tener lo que se denomina como “*miembros*”. Las Listas contienen *elementos* mientras que los Registros contienen *entradas*. Nos referiremos a aquellos valores que contienen miembros como “*contenedores*” o “*estructuras*”, y cada tanto le diremos “*propiedades*” a los miembros.



El problema

Cuando evaluamos una Lista o un Registro de forma directa, ganamos acceso a todos los miembros del valor, pero al mismo tiempo no se selecciona uno en particular.

```
CARGAR cartas con Lista 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, "J", "Q", "K"  
//Envía La representación textual de TODA La Lista "cartas", o sea:  
//(12345678910JQK)  
ENVIAR cartas
```

Esto está bien en algunos casos, pero más frecuente que no, se requiere acceder a algún miembro en específico en lugar de al contenedor completo. Existe un operador para eso.

Operador de Flecha

El operador de flecha (\rightarrow) es un operador binario cuyo operando izquierdo es el valor que contiene el miembro de interés y cuyo operando derecho es la **clave** o **índice** del miembro que se quiere acceder. La operación evalúa al valor del miembro indicado:

```
<contenedor> $\rightarrow$ <nombre> //Acceso por clave  
<contenedor> $\rightarrow$ <literal numérico> //Acceso por índice
```

Por ejemplo:

```
cartas→largo //Acceso por clave  
cartas→3    //Acceso por índice
```

El acceso por índice está intencionado para acceder los elementos de una Lista, ya que estos son indizados con Números (desde 0). Mientras tanto, el acceso por clave es de uso más general (no aplica solo a valores de Registro, expandiremos sobre esto luego).

Acceso computado

La clave o índice del miembro también pueden ser **computados**, lo cual significa que puede tomar una expresión en lugar de escribirse de forma *literal*. Para indicar un acceso computado, se encierra entre paréntesis el operando derecho, así:

```
<contenedor>→(<expresión>)
```

Ejemplo:

```
cartas→(4 + 2 * 3) //Idéntico a "cartas→10"  
cartas→("lar" + "go") //Idéntico a "cartas→Largo"  
cartas→(cartas→largo - 1) //Obtiene el último elemento de "cartas"
```

Consideraciones del acceso a miembro

Acceder elementos inexistentes en un contenedor resulta en una evaluación de Nada. Además, si el operando *izquierdo* de una operación de acceso a miembro se evalúa como Nada, se alzará un error porque Nada no tiene miembros accesibles.

Los valores de tipo Número, Texto y Lógico de hecho *pueden* ser accedidos, pero eso lo veremos después de explicar otro concepto más adelante.

Aplicaciones prácticas

Para acceder a un elemento de una Lista, se usa el índice que tiene el mismo en esta.

También, todas las Listas tienen la clave especial `largo` para evaluar la cantidad actual de elementos contenidos en la Lista:

```
CARGAR juegos con Lista "Ajedrez", "Damas", "Truco"  
  
ENVIAR juegos→0 //Envía "Ajedrez"  
ENVIAR juegos→1 //Envía "Damas"  
ENVIAR juegos→2 //Envía "Truco"  
ENVIAR juegos→largo //Envía "3"  
ENVIAR juegos→(juegos→largo - 1) //Envía "Truco"
```

Los valores guardados en un Registro pueden ser accedidos por medio de las claves definidas, y tienen una clave `tamaño` equivalente a la cantidad de entradas.

```
CARGAR juego con Registro  
nombre: "Terraria",  
género: "Acción/Aventura",  
plataformas: (Lista "PC", "Consolas", "Móvil"),  
salida: 2011,  
  
//Terraria es un juego de 2011 de Acción/Aventura para PC, Consolas, Móvil  
ENVIAR juego→nombre + " es un juego de " + juego→salida + " de " +  
juego→género + " para " + juego→plataformas→unir(", ")
```

Para acceder a los miembros del Registro `juego` se usa el operador de flecha (`→`) seguido de la clave deseada.

También podemos apreciar cómo el miembro `plataformas` contiene una Lista de 3 elementos. En `ENVIAR` nos referimos a este miembro e invocamos el método `unir` para unir con comas todos los elementos de la Lista. Detallaremos qué es un método más adelante.

Es posible acceder a todos los miembros de `plataformas` a partir de `juego`. Por ejemplo:

```
juego→plataformas→0  
juego→plataformas→1
```

Asignación a expresiones de acceso a miembro

Hasta ahora, solo habíamos visto asignaciones a variables por medio de identificadores:

CARGAR <identificador> con <expresión>

Resulta que, además de poder asignar a un identificador directo, también se puede asignar al miembro de un contenedor con una expresión de acceso a miembro, así:

CARGAR <expresión de flecha> con <expresión>

//...o sea...

CARGAR <contenedor>→<identificador> con <expresión>

CARGAR <contenedor>→(<expresión>) con <expresión>

Esto nos permite **modificar** valores de Lista o Registro, reasignando sus miembros a nuevos valores. Por ejemplo:

CARGAR unaLista con Lista 1, 2, 3

CARGAR unRegistro con Registro clave: 321, otraClave: 123

ENVIAR unaLista → 0 //Envía "1"

ENVIAR unRegistro→clave //Envía "321"

CARGAR unaLista → 0 con 42

CARGAR unRegistro→clave con 96

ENVIAR unaLista → 0 //Envía "42"

ENVIAR unRegistro→clave //Envía "96"

Nótese que si intentas hacer lo mismo con un valor primitivo, se alzará un error:

CARGAR unNúmero con 24

CARGAR unNúmero→wasd con 42

Bloques y Estructuras de Control

Profundizando el Concepto de Bloques

Previamente vimos de manera superficial qué es un bloque, con tal de poder explicar los ámbitos de variables. Básicamente, un bloque agrupa sentencias y tiene su propio ámbito.

Los indicadores de sentencia de bloque se colorean de **naranja oscuro** en el código de este documento.

Con lo que sabemos hasta ahora, podemos analizar este código:

```
CARGAR común con "Meh..."  
BLOQUE  
    CARGAR genial con "!Esto está dentro de un bloque!"  
    CARGAR común con "!!!WOW!!!"  
FIN  
//Esto envía "!!!WOW!!!", porque la variable "común" fue modificada dentro  
//de un bloque de este ámbito  
ENVIAR común  
  
//Esto envía "Nada", porque la variable "genial" fue definida en un  
//ámbito que ya no existe en este punto  
ENVIAR genial
```

La sentencia **BLOQUE** nos permite **crear y ejecutar un bloque** (o sea, una sucesión de sentencias) dentro del bloque Programa. El ámbito creado por este bloque se encuentra *dentro* del ámbito del bloque que lo ejecuta, y esto aplica para múltiples bloques anidados.

BLOQUE es un tipo de sentencia denominado “sentencia de bloque”. Los bloques se ven delimitados por una sentencia de bloque al inicio, y la sentencia **FIN** al final.

Los contenidos de bloques en este documento se desplazan hacia la derecha dependiendo de qué tan *anidados* estén, con tal de mejorar la legibilidad.

Aquí puedes ver un ejemplo de bloques anidados y cómo funciona el ámbito de variables más a detalle (lo analizaremos debajo):

```
CREAR Texto útil

BLOQUE
  BLOQUE
    CARGAR café con "colombiano"
    BLOQUE
      CARGAR algo con "No sé en dónde me metí..."
      //Envía "Me gusta mucho el café colombiano"
      ENVIAR "Me gusta mucho el café " + café
    FIN
  FIN

  //Envía "¿Nada? Eso ya no existe"
  ENVIAR "¿" + algo + "? Eso ya no existe"
  CARGAR inútil con "Esto es inútil"
  CARGAR útil con "Esto es útil"
FIN

//Envía "¿Qué es Nada? Yo solo conozco a Esto es útil"
ENVIAR "¿Qué es " + inútil + "? Yo solo conozco a " + útil

CREAR Número inútil
CARGAR inútil con 23
//Envía "23 es el vigésimo-tercer número"
ENVIAR Texto inútil + " es el vigésimo-tercer número"
```

En este ejemplo, se pueden hacer las siguientes observaciones:

- Si bien `útil` es inicializado dentro del bloque, puede ser accedido fuera de este porque fue *declarado* fuera de este. El ámbito de la variable se define en donde fue declarada, no en donde fue inicializada.
- La variable `inútil` puede ser re-declarada sin problemas al terminar el bloque, porque queda destruída cuando esto ocurre.
- Se puede acceder `café` dentro de un bloque más anidado, ya que este último se encuentra *dentro* del ámbito en el que se declaró la variable.
- La variable `algo` ya fue destruída para el punto en el que se la intenta referenciar, así que se envía el valor especial `Nada`.

Ocultación de Variables

Así como este código es válido:

```
BLOQUE
CREAR Texto var
CARGAR var con "Esto es bastante textoso."
ENVIAR var //Envía "Esto es bastante textoso"
FIN
CREAR Número var
CARGAR var con 23
ENVIAR var //Envía "23"
```

También se puede hacer algo similar con variables de mismo nombre existiendo **al mismo tiempo** en diferentes ámbitos. Por ejemplo:

```
CREAR Texto var
CARGAR var con "Esto es bastante textoso"
BLOQUE
CREAR Número var
CARGAR var con 23
ENVIAR var //Envía "23"
FIN
ENVIAR var //Envía "Esto es bastante textoso"
```

Aquí se declaran dos variables `var`. Una está en el ámbito de Programa, siendo un Texto; y otra está en el ámbito de otro bloque, siendo un Número. Esto nos permite declarar varias variables bajo el mismo identificador simultáneamente, lo cuál nos será útil más adelante cuando nuestro código crezca en complejidad.

La declaración anidada se hace de forma explícita, con `CREAR`. Nótese que la variable en el ámbito más anidado **oculta** la del ámbito más general hasta que el anidado termina.

En otras palabras, las variables de ámbitos más anidados tienen mayor prioridad en caso de un conflicto de nombres.

La sentencia `BLOQUE` por sí sola no es muy útil. Existen otras sentencias de bloque que alteran el flujo del programa. Estas se conocen como **estructuras de control**.

Estructuras de Control

Las sentencias de estructura de control se colorean de **naranja oscuro** en el código de este documento. **Todas** las sentencias de estructura de control *crean un bloque implícitamente* que debe terminarse con **FIN**.

A continuación, veremos los siguientes temas:

- [Estructuras Condicionales](#)
 - [SI](#)
 - [SI...SINO](#)
- [Expresiones Lógicas y Operadores Lógicos](#)
- [Estructuras Iterativas](#)
 - [REPETIR](#)
 - [MIENTRAS](#)
 - [HACER...HASTA](#)
 - [PARA Corto](#)
 - [PARA Largo](#)
- [Otras Sentencias](#)
 - [TERMINAR](#)
 - [PARA CADA](#)

Estructuras Condicionales

↑ Luego de la Sentencia BLOQUE, estas sentencias de bloque son las segundas más básicas. Estas permiten un amplio grado de interactividad en el Tubérculo.

Sentencia SI

↑ La Sentencia SI evalúa una expresión a su *representación lógica*. Si la misma evalúa a Verdadero, ejecuta su bloque. Si no, lo ignora.

```
SI Falso
    //Las sentencias escritas aquí dentro se IGNORARÁN
FIN

SI Verdadero
    //Las sentencias escritas aquí dentro se EJECUTARÁN
FIN
```

Sentencia SINO y la estructura SI...SINO

↑ La Sentencia SINO siempre va luego de un bloque SI, reemplazando la sentencia FIN que lo delimitaría, y define un bloque propio que se ejecutará **solo si** dicho bloque previo no se ejecuta. Esto crea una forma “SI...SINO”, que se puede leer como “Si esto es verdad, haz esto. Si no, haz esto otro”.

Expandiendo del ejemplo anterior, quedaría así:

```
SI Falso
    //Las sentencias escritas aquí dentro se ignorarán
SINO
    //En cambio, esto se ejecutará
FIN

SI Verdadero
    //Las sentencias escritas aquí dentro se ejecutarán
SINO
    //En cambio, esto se ignorará
FIN
```

También, puedes especificar otra condición inmediatamente después del SINO:

```
SI 2 es 4
    //Esto no se ejecuta, porque 2 no es igual a 4
SINO SI 2 es 3
    //Esto no se ejecuta porque 2 tampoco es 3
SINO SI 2 es 2
    //Esto se ejecuta, porque 2 es 2
SINO
    //Esto no se ejecuta, porque ya se ejecutó otro bloque
FIN
```

Es importante reconocer que, cuando se ejecuta un bloque SINO, es porque la o las condiciones de SI anteriores no se cumplen. Esto nos ayuda a deducir el valor o rango de valores de una variable según el fragmento de código que se hable. Por ejemplo:

```
SI núm excede 0
    //Si este bloque se ejecuta, el número es positivo
SINO SI núm precede 0
    //Si este bloque se ejecuta, el número es negativo
SINO
    //Por descarte, si este bloque se ejecuta, "núm" es 0
FIN
```

```
SI núm precede 0
    //núm es negativo aquí
SINO
    //núm es positivo o cero aquí
    SI núm precede 5
        // "núm" está entre 0 y 4 aquí
    FIN
FIN
```

En la siguiente sección estudiaremos los **operadores lógicos** así que no te preocupes si no entiendes qué es excede, precede, es, etc.

Cabe mencionar que estos son ejemplos básicos. Es importante comprender esta lógica para resolver problemas más complejos. Puedes revisitar estas páginas luego de la siguiente sección si sientes que no entendiste bien.

Operadores Lógicos

Previamente vimos unos cuantos tipos de operadores. Ahora veremos unos operadores un poco más enfocados a la toma de decisiones.

↑ Una expresión lógica es aquella que evalúa a un valor Lógico: Verdadero o Falso. Hay tres subtipos de estos operadores:

- [Operador de Negación Lógica](#)
- [Operadores Relacionales](#)
- [Operadores Conectores](#)

Operador NO

↑ Representa una negación lógica. Es un operador prefijo y similar al prefijo aritmético `-`.

Convierte el valor evaluado a su *representación lógica* y lo **invierte**. O sea que si la expresión se convierte inicialmente a Verdadero, la evaluación final es Falso (y vice-versa):

```
SI no Falso
    //"no Falso" es Verdadero, así que esto se ejecuta
FIN

SI no Verdadero
    //"no Verdadero" es Falso, así que esto se ignora
FIN

SI no 3
    //"no 3" es lo mismo que "no Verdadero"
FIN

SI no Nada
    //"no Nada" es lo mismo que "no Falso"
FIN
```

Operadores relacionales

↑ Un operador relacional es aquel que define una relación entre dos operandos y resulta en un valor Lógico basado en si esta se cumple (Verdadero) o no (Falso).

Existen tres variantes de operadores relacionales:

- Operadores de **Igualdad/Desigualdad**

```
SI 2 no es 4
    //Esto se ejecuta, porque 2 no es igual a 4
FIN
```

- Operadores de **Similitud**

```
SI 2 parece "2"
    //Esto se ejecuta, porque 2 es parecido a "2"
FIN
```

- Operadores de **Comparación u Ordenamiento**

```
SI 2 excede 4
    //Esto no se ejecuta, 2 no es mayor que 4
FIN
```

Los operandos de una operación de igualdad o similitud pueden ser de cualquier tipo. Los de una comparación *deberían* ser Números o Textos idealmente, pero se permiten otro tipo de comparaciones.

Los números se comparan según su valor numérico. Ese es obvio.

Los Textos se comparan según orden lexicográfico, lo cual es algo como el orden alfabético pero extendido a dígitos y símbolos, según el formato Unicode (ej.: “a” < “b”).

Para los Lógicos... Verdadero siempre excede a Falso y Falso siempre precede a Verdadero.

Otros tipos igual se pueden comparar pero no vamos a cubrir eso aquí.

A modo de ejemplo, considera las siguientes variables:

```
CARGAR a con 2
CARGAR b con 4
```

...y revisa la tabla de operadores:

Operador	Evaluación	Ejemplos de evaluación...	
		Verdadero	Falso
<code>es</code>	Verdadero si los operandos son iguales, tanto en tipo como en valor.	<code>a es 2</code> <code>2 es 2</code> <code>"hola" es "hola"</code>	<code>a es 4</code> <code>2 es "2"</code>
<code>no es</code>	Verdadero si los operandos no son iguales, tanto en tipo como en valor.	<code>a no es "2"</code> <code>2 no es "2"</code> <code>"x" no es "y"</code>	<code>a no es 2</code> <code>a no es a</code> <code>"m" no es "m"</code>
<code>parece</code>	Verdadero si los operandos son iguales en valor. Incluye conversión de tipo.	<code>a parece 2</code> <code>a parece "2"</code> <code>2 parece "2"</code>	<code>a parece b</code> <code>"x" parece "y"</code>
<code>no parece</code>	Verdadero si los operandos no son iguales en valor. Incluye conversión de tipo.	<code>a no parece 4</code> <code>a no parece "2"</code>	<code>a no parece "2"</code> <code>2 no parece "2"</code>
<code>excede</code>	Verdadero si el operando de la izquierda vale más que el de la derecha.	<code>b excede a</code> <code>"12" excede 2</code>	<code>a excede b</code> <code>2 excede 2</code>
<code>no excede</code>	Verdadero si el operando de la izquierda vale menos o igual que el de la derecha.	<code>a no excede b</code> <code>2 no excede 2</code>	<code>"4" no excede 2</code>
<code>precede</code>	Verdadero si el operando de la izquierda vale menos que el de la derecha.	<code>a precede b</code> <code>"a" precede "b"</code>	<code>b precede a</code> <code>3 precede 3</code> <code>"z" precede "a"</code>
<code>no precede</code>	Verdadero si el operando de la izquierda vale más o igual que el de la derecha.	<code>b no precede a</code> <code>a no precede 2</code>	<code>a no precede b</code>

Esto explica brevemente el funcionamiento de cada operador relacional. Puedes aprender más en la [Referencia](#).

Operadores conectores

↑ Los operadores conectores lógicos permiten conectar múltiples expresiones. Esto principalmente se aplica a expresiones lógicas, pero puede usarse cualquier expresión. Los operadores listados aquí evalúan al valor de uno de sus dos operandos, que *pueden o no* ser de tipo Lógico.

Los conectores lógicos disponibles son:

- Operador de **Conjunción Lógica**: `y`
- Operador de **Disyunción Lógica**: `o`

Operador	Uso	Descripción
<code>y</code>	<code>a y b</code>	Devuelve el operando <code>a</code> si este evalúa a Falso. De lo contrario, devuelve <code>b</code> . Si se usa con Lógicos, hace que se devuelva Verdadero si ambos operandos son Verdadero. De lo contrario, devuelve Falso.
<code>o</code>	<code>a o b</code>	Devuelve el operando <code>a</code> si este evalúa a Verdadero. De lo contrario, devuelve <code>b</code> . Si se usa con Lógicos, hace que se devuelva Verdadero si cualquier operando es Verdadero. De lo contrario, devuelve Falso.

Ahora veamos unos ejemplos para que tenga más sentido:

```
CARGAR v1 con Verdadero y Verdadero //Asigna Verdadero
CARGAR v2 con Verdadero y Falso //Asigna Falso
CARGAR v3 con Falso y Verdadero //Asigna Falso
CARGAR v4 con Falso y Falso //Asigna Falso
CARGAR v5 con Verdadero y (2 es 2) //Asigna Verdadero
CARGAR v6 con Verdadero y (3 es 2) //Asigna Falso
CARGAR v7 con "Té" y "Café" //Asigna "Café"
CARGAR v8 con Falso y "Té" //Asigna Falso
CARGAR v9 con "Té" y Falso //Asigna Falso
```

Evaluación de cortocircuito

Como las operaciones de conectores lógicos se evalúan de izquierda a derecha, se comprueban posibles evaluaciones de “cortocircuito” según las siguientes reglas:

- Falso y algo evalúa en cortocircuito hacia Falso.
- Verdadero o algo evalúa en cortocircuito hacia Verdadero.

Nótese que algo **nunca se evalúa** en las dos expresiones. Se ignora por cortocircuito.

Demostración:

```
CARGAR a con Falso o Verdadero //Asigna Verdadero
CARGAR b con Falso o 42 //Asigna 42
CARGAR c con Verdadero y 32 //Asigna 32
CARGAR d con Nada o Verdadero //Asigna Verdadero
CARGAR e con Nada y Falso //Asigna Nada
CARGAR f con Falso o (2 y 0) //Asigna 0
CARGAR g con (0 y 1) no es (2 o 4) //Asigna Verdadero
CARGAR h con "Té" es Falso //Asigna Falso
CARGAR i con "Café" o Falso //Asigna "Café"
CARGAR j con (2 no es 2) o (2 es 2) //Asigna Verdadero
```

Igualdad y similitud

En PuréScript, la *igualdad* (`es` / `no es`) y la *similitud* (`parece` / `no parece`) son las dos formas de definir una relación de paridad entre dos valores. Se dice que:

- **Todos** los valores *iguales* entre sí también son *similares* entre sí.
- **Algunos** valores *similares* entre sí también son *iguales* entre sí.

Dados dos valores, A y B, se siguen estos pasos en orden para concluir si son **iguales**:

1. Si los tipos de A y B no coinciden, los valores **no son iguales**.
2. Si se cumple la condición correspondiente al tipo de los valores, son **iguales**:
 - **Números** — coinciden en signo y magnitud.
 - **Textos** — coinciden en cada carácter, incluso en mayúsculas y tildes.
 - **Lógicos** — **no** son opuestos.
 - **Otro** — son exactamente el mismo valor **internamente**.

Dados dos valores, A y B, se siguen estos pasos en orden hasta concluir si son **similares**:

1. Si los tipos de A y B coinciden, **se verifica si los valores son iguales**.
2. Si uno de A o B es de tipo Marco o Función, los valores **no son similares**.
3. Si A es `Nada` y B **no** es un primitivo, o vice-versa, los valores **no son similares**.
4. Si *uno* de A o B es de tipo Texto, Lista o Registro, se **convierte** el otro a ese tipo.
5. Si *uno* de A o B es de tipo Número, se **convierte** el otro a *Número*.
6. Si *uno* de A o B es de tipo Lógico, se **convierte** el otro a *Lógico*.
7. Se evalúa si A y B son **similares** según si los *valores convertidos* son **iguales**.

Precedencia de operadores lógicos

Los operadores relacionales, de negación y conectores se usan juntos a menudo para bloques condicionales, como por ejemplo:

```
CARGAR valor con 3  
SI valor no precede 0 y valor no excede 9  
    ENVIAR "Eso es solo un dígito"  
SINO  
    ENVIAR "Esto es cualquier número que no sea de un dígito"  
FIN
```

La precedencia de los mismos es: `no > (excede/precede) > (es/parece) > y > o`

Puedes encontrar una mejor explicación de precedencias junto a la tabla de precedencias completa con todo el contexto en la [Referencia](#).

Estructuras Iterativas

- ↑ La mayoría de estructuras iterativas siguen la misma lógica que las estructuras condicionales, y son esencialmente bloques que se repiten (*iteran*) según una condición.

Sentencia REPETIR

- ↑ Esta es la sentencia iterativa más fácil de entender. Ejecuta un bloque la cantidad de veces que especifiques. Requiere la palabra clave `veces` luego de la condición.

El siguiente ejemplo demuestra la sintaxis. Este código envía los números del 1 al 10:

```
CARGAR ejecuciones con 1
REPETIR 10 veces
    //Envía el valor actual de "ejecuciones" y le suma 1
    ENVIAR ejecuciones
    SUMAR ejecuciones
FIN
```

El bloque de la sentencia es ejecutado múltiples veces, hasta que se satisface la condición de cantidad de repeticiones.

Sentencia MIENTRAS

- ↑ Esta sentencia, como su nombre lo indica, ejecuta su bloque reiteradas veces **mientras** que la condición evalúe a Verdadero.

La sintaxis es idéntica a la sentencia SI. Esto se ve así:

```
CARGAR ejecuciones con 100
MIENTRAS ejecuciones excede 0
    //Esto se va a ejecutar 100 veces
    //En cada ejecución, se restará 1 a "ejecuciones"
    RESTAR ejecuciones
FIN
```

El cuerpo de la sentencia se repite hasta que su condición deja de cumplirse.

Bucles Infinitos y Límites a considerar

La condición de iteración de una estructura iterativa debería dejar de cumplirse eventualmente, debido a algún cambio de estado en alguna de las iteraciones.

Si la condición de iteración no se deja de cumplir eventualmente, el bloque se ejecutaría infinitamente, lo cual **alzará un error** luego de una gran cantidad de ejecuciones.

Esto es debido a una serie de reglas contra abusos:

- Una instancia de ejecución de un **Tubérculo** está limitada a **1000** “créditos”.
- Los créditos se reducen en **1** por cada **sentencia** evaluada.
- Los créditos se reducen en **0.1** por cada **expresión** evaluada.
- Esta cantidad de créditos generalmente es más que suficiente, pero si se supera este límite, **se alzará un error** para proteger a Bot.

No te preocupes. La forma más común de golpear el límite de créditos es con un **bucle infinito**, así que no es realmente un límite creativo. ¡Si este límite *no* existiera, los bucles infinitos harían que Bot quede **inoperante** por un buen rato!

Además, tienes que considerar las limitaciones que están más relacionadas a Discord.

Un **Tubérculo** solo puede desencadenar 1 envío de mensaje. Si bien usar ENVIAR repetidas veces hará que se acumulen todos los envíos en uno solo, debes considerar también los límites de mensajes de Discord. Cosas como el límite de caracteres, límite de Marcos, etc.

Sentencias HACER y HASTA, y la estructura HACER...HASTA

↑ El funcionamiento de la estructura es similar al de MIENTRAS. Una diferencia importante es que el bloque de HACER...HASTA **siempre** se va a ejecutar *al menos una vez* (incluso si la condición de iteración no se satisface). Esto se debe a que se evalúa la condición *luego* de ejecutar el bloque. Otra diferencia a tener en cuenta es que se evalúa el **opuesto** de la expresión dada. Por ende: el bloque se ejecuta *hasta* que la condición evalúa a Verdadero.

También, en lugar de señalizar el fin del bloque con FIN, se usa HASTA seguida de la expresión condicional a evaluar.

Esto tendrá más sentido si lo vemos en código. Veamos el siguiente ejemplo:

```
CARGAR ejecuciones con 100

HACER
    //Esto se va a ejecutar 100 veces
    //En cada ejecución, se restará 1 a "ejecuciones"
    RESTAR ejecuciones
HASTA ejecuciones no excede 0

//Estamos seguros de que "ejecuciones" vale 0 en este punto

HACER
    //Esto se va a ejecutar 1 vez, incluso si "ejecuciones" ya no excede 0
    RESTAR ejecuciones
HASTA ejecuciones no excede 0

//En este punto, "ejecuciones" vale -1
```

Básicamente, primero ejecutamos el contenido de los bloques iterativos y **después** comprobamos que la condición siga dando Falso. Pequeños pero importantes cambios.

Sentencia PARA Corta

↑ La Sentencia PARA tiene **2** versiones dependiendo de la sintaxis utilizada: *Corta* y *Larga*.

La Sentencia PARA Corta acepta un identificador para declarar una variable totalmente **local** a la **sentencia** (“ámbito de sentencia”, lo vimos antes), y un **rango de valores** por los cuales la variable deberá *desplazarse* paso a paso entre cada iteración. Como la variable pertenece al PARA, será destruída cuando el mismo termine.

Primero se especifica el identificador a declarar y luego el rango de valores que tomará la variable bajo el identificador indicado. Para el rango de valores se indica el **valor inicial** con **desde** y el **valor destino** con **hasta**. Ambos extremos del rango son *inclusivos* y se **convierten** a su *representación numérica* si no lo son.

Ejemplo:

```
//Envía los siguientes valores: "5", "6" "7", "8", "9", "10"
PARA ejecución desde 5 hasta 10
    ENVIAR ejecución
FIN

//Envía los siguientes valores: "7", "6" "5", "4", "3"
PARA ejecución desde 7 hasta 3
    ENVIAR ejecución
FIN

//Envía "Nada"
ENVIAR ejecución
```

Nótese cómo la *dirección* hacia la cual se desplaza el valor por iteración se determina según si el **valor destino** es mayor (incrementa) o menor (decrementa) que el **valor inicial**. El desplazamiento siempre se realiza en intervalos de **±1**.

La Sentencia PARA Corta es ideal para manejar iteraciones con una *variable contador* de forma sencilla. La variable que se inicializa es totalmente local al PARA, y por ende es destruída al terminar de iterar.

Sentencia PARA Larga

↑ La Sentencia PARA Larga también declara una variable totalmente **local** a la sentencia. En lugar de solo un identificador y un rango de valores, la contraparte Larga toma **3 pasos** antes de los contenidos del bloque:

1. Un paso de **inicialización** — Se realiza una **única** vez *antes* de la **comprobación**.
Consiste de un identificador, la palabra clave **con** y una **expresión**. La expresión se usa para inicializar el identificador dentro de esta sentencia.
2. Una **expresión de comprobación** — Se evalúa siempre *antes* de considerar ejecutar el bloque iterativo. Se precede por un **MIENTRAS** que no tiene **ningún efecto**, y puede ser cualquier **expresión**, ya que será **convertida** a Lógico de todas formas.
Determina si se debe **ejecutar** el bloque iterativo al evaluarse como **Verdadero**, o si **parar** al evaluarse como **Falso**.
3. Una **sentencia de actualización** — Se evalúa **siempre después** de haber ejecutado el bloque iterativo. Puede ser cualquier **sentencia** ().

Pongámoslo en práctica:

```
PARA ejecución con 100 MIENTRAS ejecución excede 0 RESTAR ejecución
    //Esto se va a ejecutar 100 veces
    //En cada ejecución, se restará 1 a "ejecución"
    FIN

PARA i con 1 MIENTRAS i precede 200 MULTIPLICAR i con 2
    //Esto se va a ejecutar 8 veces
    //En cada ejecución, se multiplicará "ejecución" por 2
    FIN
```

Esta variante de PARA también es ideal para manejar iteraciones con una *variable contador*, pero solo la usarías por encima de la contraparte *Corta* si tienes alguna **condición de iteración** muy particular o si quieres incrementar/decrementar la variable por intervalos diferentes de ±1.

Fíjate bien porque también puede que quieras usar un **MIENTRAS** en lugar de un **PARA Largo**. La diferencia entre usar **MIENTRAS** con una variable propia y un **PARA Largo** es que la variable del **PARA** solo vive dentro de esta sentencia. ¡A las herramientas hay que usarlas para el trabajo correcto!

Otras Sentencias de Control

Sentencia TERMINAR

↑ La Sentencia **TERMINAR** interrumpe la ejecución del bloque iterativo actual y termina la repetición del mismo. Esto se puede usar también para terminar prematuramente la ejecución del Programa o una *Función* (veremos funciones [más abajo](#)).

```
MIENTRAS Verdadero
    //Esto solo se ejecutará 1 vez, debido a TERMINAR
    TERMINAR
FIN
```

Sentencia PARA CADA

↑ En el caso de querer iterar sobre los elementos de un contenedor, se puede emplear la Sentencia **PARA CADA** en conjunto con la palabra clave **en** para formar la siguiente sintaxis:

```
CREAR Lista gnomos
CARGAR gnomos con Lista "Sinhik", "Traybar", "Umnam", "Grawin"

PARA CADA gnomo en gnomos
    //Envía este mensaje con el nombre de todos Los gnomos
    ENVIAR "Este gnomo se llama " + gnomo + ", buen tipo."
FIN
```

En este ejemplo, se declara la variable **gnomo** en cada iteración para representar al elemento de **gnomos** que se esté iterando actualmente.

PARA CADA te permite iterar un bloque por cada miembro de un **contenedor**, dando un identificador directo para el miembro de la iteración actual. En esta estructura, simplemente especificas el **contenedor** (**gnomos**) y cómo quieres referirte a cada miembro por iteración (**gnomo**).

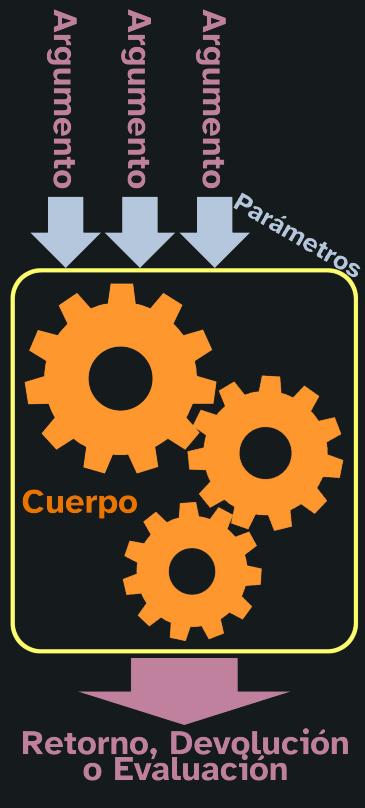
Si quieres realizar una operación larga sobre cada elemento de una **Lista** y no te importa el índice, o si necesitas aplicar una operación a todas las entradas de un **Registro**, esta es la mejor opción que hemos visto para ello.

Funciones

Las funciones son un *tipo de valor* especial y un concepto esencial. Una Función en PuréScript es básicamente un bloque reutilizable. Esto significa que es un conjunto de instrucciones que pueden ser llamadas o invocadas. Al ser llamadas, se ejecutan y devuelven un valor, también pueden modificar variables externas. Puedes pasárselas valores como argumentos al llamarlas.

Recuerda que una Función es un tipo de **valor**, como lo sería cualquier Número, Texto, etc. Por ende, puede almacenarse en una variable y respeta el concepto de ámbito de variable.

Los identificadores que se refieran a valores de Función se colorearán de **amarillo** en el código de este documento.



Retorno, Devolución
o Evaluación

Demostración ilustrativa
de una Función

Expresión de Llamado

Previamente habíamos mencionado que las expresiones **no** tienen *efectos secundarios*. Sin embargo, **existe una excepción a esto**.

Puedes *llamar* una Función usando una **expresión de llamado**. Esta consiste de unos paréntesis inmediatamente después de otra expresión que evalúe a una Función:

```
<función>()
```

La expresión antes de los paréntesis suele ser simplemente una variable de Función.

Llamar a una Función hace que la misma se ejecute. La *evaluación* de una expresión de llamado es el *valor devuelto* por la Función llamada al terminar de ejecutarse. Por eso mismo se dice que se devuelve o retorna.

PuréScript cuenta con varias funciones **ya listas para usarse**. Entre ellas, existe una llamada `dado`, que *devuelve* un número al azar entre 1 y 6 (como si fuera un dado de verdad) cuando se la llama/ejecuta:

```
ENVIAR dado() //Envía un número aleatorio entre 1 y 6
```

Argumentos de Función

Las Funciones pueden tener *parámetros*, que requieren que les pases *argumentos* para darles valor durante su ejecución. Un parámetro es básicamente **una variable que recibe un valor externo** (llamado **argumento**) y es **usado por la Función durante su ejecución**. Los parámetros de la Función reciben los argumentos *antes* de ejecutarla.

En una expresión de llamado, puedes especificar los argumentos que quieras pasar entre paréntesis, separados por coma. Por ejemplo, para llamar una función con N argumentos:

```
<función>(<argumento1>, <argumento2>, (...), <argumentoN>)
```

Entonces, si tuviéramos una Función `sumar` que calcule y devuelva el resultado de una suma de 2 parámetros, podríamos llamarla pasando 2 argumentos de la siguiente forma:

```
ENVIAR sumar(2, 2)    //Envía "4"  
ENVIAR sumar(4, 1)    //Envía "5"  
ENVIAR sumar(10, -2)  //Envía "8"
```

En conclusión:

- Las funciones pueden tener **parámetros**.
- Los *parámetros* son solo una **variable** más durante la ejecución de la Función.
- Al *llamar* funciones con parámetros, se pasa un **argumento** a cada *parámetro*.
- Los *parámetros* **reciben** los valores de los *argumentos* pasados al llamar la Función.

Sentencia EJECUTAR

La Sentencia **EJECUTAR** en general solo es útil para el tema que estamos viendo ahora. **EJECUTAR** simplemente evalúa una expresión. ¡Eso es todo! Esta es la sintaxis:

```
EJECUTAR <expresión>
```

Esto sería inútil si ninguna expresión tuviera efectos secundarios como habíamos visto hasta ahora. Sin embargo, con expresiones de llamado, **podemos usar esta sentencia para ejecutar funciones** y olvidarnos del *valor de retorno* (valor devuelto por la Función):

```
EJECUTAR unaFunción()
```

Nótese que la Función del ejemplo debería tener algún efecto secundario al llamarla para que la sentencia siquiera haga algo.

Clasificación de Funciones

Existen múltiples formas de clasificar funciones. Por ahora, vamos a limitarnos a clasificarlas según su propósito y operatividad.

Una Función puede ser pura, impura o directamente un procedimiento:

- **Funciones puras** — son aquellas que **no** tienen efectos secundarios y que ante la misma serie de argumentos producen los mismos resultados. **Devuelven** un valor.
- **Funciones impuras** — son aquellas que **pueden** tener efectos secundarios y **devuelven** un valor que no depende solo de la serie de argumentos especificada.
Ejemplo – `dado()`
- **Procedimiento** — son aquellas que **siempre** devuelven Nada. Su único propósito es causar *efectos secundarios* al ejecutarse.

Expresión de Función

Ahora que sabemos llamar Funciones, podemos intentar *crearlas* nosotros mismos. Al igual que los tipos de valor vistos hasta ahora, las funciones tienen una expresión literal. Esto nos permite definir nuestra Función como queramos y almacenarla en una variable.

Una *expresión de Función* comienza con el indicador de tipo `Función`, seguido de los identificadores de sus parámetros separados por comas y entre paréntesis, seguido del `cuerpo` de la Función terminado con la sentencia `FIN`.

Esta expresión **no** ejecuta la Función creada, para eso se usan las *expresiones de llamado* que se mencionaron anteriormente.

```
Función(<argumento1>, <argumento2>, (...), <argumentoN>)
    ...sentencias/cuerpo de Función
FIN
```

Revisemos este ejemplo sencillo:

```
CARGAR saludo con Función()
    ENVIAR ";Hola mundo!"
FIN

EJECUTAR saludo() //Envía ";Hola mundo!"
```

Aquí creamos una Función cuyo cuerpo tiene una sola sentencia: `ENVIAR ";Hola mundo!"`, y la asignamos a la variable `saludo`. A continuación, en el programa principal, usamos la sentencia `EJECUTAR` para llamar dicha Función, lo cual resulta en que se envíe un mensaje.

Parámetrosopcionales

PuréScript te permite ingresar tantos argumentos como quieras en un llamado de Función. Cuando ingresas **menos** argumentos que los que te pide la declaración de Función, **se alza un error**, ya que la ejecución con parámetros no inicializados está **prohibida**.

Los **parámetrosopcionales** te permiten asignar un valor por defecto a un parámetro que no se le pasó un argumento. Esto hace a tus funciones más fáciles de planear y llamar.

Primero veamos un ejemplo de una Función *sin* parámetros opcionales:

CARGAR producto con Función(a, b)
DEVOLVER a * b
FIN

En la anterior Función, si no se pasan `a` o `b`, se alza un error. Eso puede estar bien dependiendo del objetivo que se tenga en mente, pero podemos usar valores por defecto para definir parámetros opcionales:

```
CARGAR producto con Función(a, b: 1)
    DEVOLVER a * b
FIN

ENVIAR producto(2) //Envía "2", porque 2 * 1 = 2
```

Aquí, si al llamar `producto` no se pasa un valor para `b`, tomará el valor `1`. No pasar a seguirá arrojando un error dado que no se le especificó como parámetro opcional.

Retorno de Valores

Como ya se ha mencionado, las funciones *devuelven* o *retornan* un valor, conocido como *valor de retorno*.

El valor de retorno es el resultado de la evaluación de una *expresión de llamado*, así que puede considerarse un mensaje que emite la Función al exterior. Este comportamiento se logra con la Sentencia [DEVOLVER](#).

DEVOLVER <expresión>

La expresión indicada en la Sentencia DEVOLVER será devuelta como resultado de evaluación del llamado de la Función en la que se encuentra.

Para que tenga más sentido, veamos cómo podríamos implementar la Función `sumar` que ejemplificamos antes con estos nuevos conocimientos:

```
CARGAR sumar con Función(a, b)
    DEVOLVER a + b
FIN

ENVIAR sumar(2, 2)    //Envía "4"
ENVIAR sumar(4, 1)    //Envía "5"
ENVIAR sumar(10, -2)  //Envía "8"
```

Es importante mencionar que cuando se llama `DEVOLVER` en una Función, la misma **deja de ejecutarse ahí mismo**, devolviendo el valor *inmediatamente* en lugar de ejecutar las siguientes sentencias (si hubieran).

Como de momento las Funciones que hemos visto no son tan largas, puede no tener mucho sentido crearlas. Obviamente no tiene sentido crear una Función `sumar`, pero el tema escalará en complejidad de a poco y se mostrarán algunas Funciones bastante largas que justificarán su uso en breve.

Analicemos un ejemplo un poco más completo:

```
CARGAR esPar con Función(núm)
    SI núm % 2 es 0
        DEVOLVER Verdadero
    SINo
        DEVOLVER Falso
    FIN

    //Nunca se Llega a este punto debido a los DEVOLVER de arriba...
FIN

ENVIAR esPar(42) //Envía "Verdadero"
ENVIAR esPar(11) //Envía "Falso"
ENVIAR esPar(38) //Envía "Verdadero"
ENVIAR esPar(23) //Envía "Falso"
```

La función `esPar` toma un argumento `núm` y devuelve `Verdadero` si `núm` es un Número par, `Falso` de lo contrario.

Paso de Parámetros por Valor y por Referencia

Cuando pasas un valor primitivo a una Función, se realiza una “copia” del valor en el ámbito de la misma. Debido a la *ocultación de variables*, el nombre en el ámbito de la Función toma prioridad sobre nombres idénticos en ámbitos más generales. Modificar esta copia en el cuerpo de la Función no afectará al original incluso si se llaman igual:

```
CARGAR fn con Función(var)
    //La variable "var" aquí es una copia del valor ingresado
    CARGAR var con 30
FIN

CARGAR var con 15

//Se ejecuta la función con una copia de "var"...
EJECUTAR fn(var)

//Envía 15, porque "var" de este ámbito nunca fue modificada
ENVIAR var
```

Esto se conoce como “paso de parámetros **por valor**”. Caso en el cual las modificaciones del parámetro durante la ejecución de la Función no afectan al argumento original.

Veamos qué ocurre al pasar una estructura (como una Lista o un Registro) a una Función:

```
CARGAR agregarGorila con Función(li)
    //EXTENDER agrega un nuevo elemento al final de una Lista
    EXTENDER li con "Gorila"
FIN

CARGAR primates con Lista "Mono", "Orangután", "Lémur"

//Envía "Mono, Orangután, Lémur"
ENVIAR primates→unir(", ")

//Se envía una referencia al valor de Lista de "primates"...
EJECUTAR agregarGorila(primates)

//Envía "Mono, Orangután, Lémur, Gorila"
ENVIAR primates→unir(", ")
```

Pasar una variable **por referencia** hace que, si la variable se modifica en la Función, el cambio se vea reflejado fuera de la misma (efecto secundario).

Nótese que **reasignar** el parámetro `l1` dentro de la anterior Función a otro valor **no modificaría** el argumento, sino que solo le da otro valor al *parámetro*, lo cual *perdería el vínculo* con el valor original (EXTENDER en cambio **modifica** un valor).

También, es importante discriminar una *asignación directa* de una *asignación de flecha*: ante el código `CARGAR l1→0 con "Gato"`, **sí se vería modificado** `l1` debido a que se reemplazó el valor de *uno de sus miembros*, *no el valor de l1 en sí*. Por ende, la expresión `l1→0` fuera de la Función también se evaluaría como `"Gato"`.

Siempre ten en cuenta en estos casos: **asignar no es lo mismo que modificar**.

Entonces:

- **Datos primitivos:** se pasan por valor. "Parámetros como copias"
- **Estructuras:** se pasan por referencia. "Parámetro y argumento vinculados entre sí"

Ahora... ¿quieres saber un secreto? En realidad los valores de estructura también se pasan como copias a las Funciones. *Todos los argumentos se copian al pasarse a un parámetro*.

Entonces... ¿por qué los cambios a primitivos nunca se ven reflejados en el exterior? Bueno, es más de lo que veníamos hablando: **asignar no es lo mismo que modificar**. Los valores primitivos **no pueden modificarse, solo pueden asignarse**. O sea, es **imposible** que se refleje un cambio en un argumento primitivo.

Fíjate que la Sentencia EXTENDER no tiene efecto alguno en Números ni Textos ni Lógicos, y que los valores primitivos en general no tienen miembros asignables. Este es el motivo fundamental por el cual se crea esta diferencia

...igual es más fácil simplemente recordar que los primitivos se pasan como copias y las estructuras como referencia, ¿no?

Solo ten cuidado. Es importante saber este concepto para llevarse menos sorpresas al trabajar con funciones.

Recursividad

Una Función puede llamarse a sí misma en su cuerpo. Este concepto se conoce como “*recursividad*”. Plantean un comportamiento similar al de un bloque iterativo, esencialmente repitiendo la ejecución su cuerpo una y otra vez en una *sucesión de llamadas* con diferentes estados hasta que uno o más factores causan que la Función deje de llamarse a sí misma (si se llamara infinitamente, eventualmente **alzaría un error**).

```
CARGAR factorialDe con Función(núm)
    SI núm excede 1
        DEVOLVER núm * factorialDe(núm - 1)
    SINO
        DEVOLVER núm
    FIN
FIN

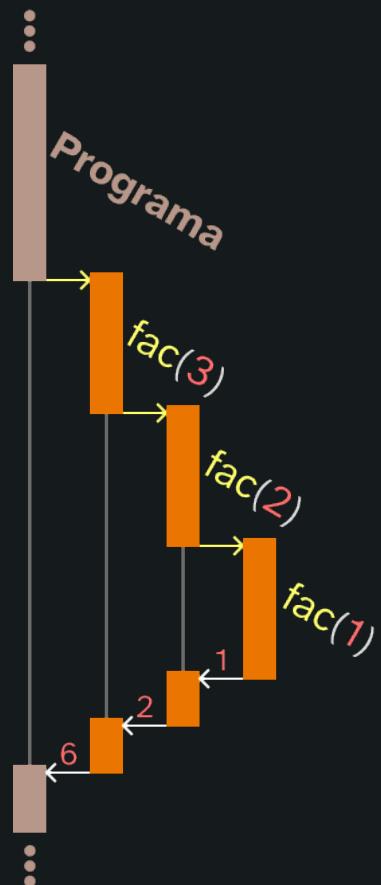
ENVIAR factorialDe(2) //2! = 2 * 1 = 2
ENVIAR factorialDe(4) //4! = 4 * 3 * 2 * 1 = 24
ENVIAR factorialDe(6) //6! = 6 * 5 * 4 * 3 * 2 * 1 = 720
```

Cada llamado de Función se agrega a una “pila de ejecución”. Cuando una Función llama a otra, **pospone el resto de sentencias en su ejecución** hasta que termine de evaluarse la Función que llamó (porque *no se puede continuar* hasta no tener un resultado de evaluación, incluso si el mismo es Nada).

Esto también aplica cuando una Función se llama a sí misma, y en general cuando se llama una Función desde cualquier parte.

Cuando expresamos `factorialDe(6)`, en algún punto de la ejecución vamos a tener 5 funciones en fila esperando que otra Función termine de evaluarse.

Es bastante literal lo de “pila de ejecución”. Es como apilar funciones una encima de la otra: **la primera función en ser llamada es la última en concluirse**.



Demostración ilustrativa de la pila de ejecución con una Función recursiva

Recursividad vs. Iteración

Si bien la recursión puede ser idéntica a un bloque iterativo en algunos casos, hay ocasiones en las que conviene usar una Función recursiva antes que lo segundo:

```
CARGAR recorrer con Función(nodo)
  SI no esRegistro(nodo)
    TERMINAR
  FIN

  //Aquí harías algo con el nodo...

  PARA CADA hijo en nodo→valores()
    EJECUTAR recorrer(hijo)
  FIN
FIN
```

Esta Función permite recorrer un árbol de Registros (Registros que contienen Registros que contienen Registros y así, formando una forma similar a la de un árbol). Los nodos del árbol se recorren de manera recursiva, realizando la acción entre medio con cada uno.

Este ejemplo es más fácil de expresar con recursión que con solo estructuras iterativas. Solo por si gustas, así se vería el código de arriba reescrito de forma iterativa:

```
SI no esRegistro(nodo)
  TERMINAR
FIN

CARGAR pila con nodo→valores() //Llenar pila

MIENTRAS no pila→vacía()
  CARGAR nodo con pila→robarÚltimo() //Sacar de pila

  //Aquí harías algo con el nodo...

  PARA CADA hijo en nodo
    EXTENDER pila con hijo //Colocar en pila
  FIN
FIN
```

Como verás, es más complicado de seguir y un poco más largo que la versión recursiva. Esto se debe a que con la versión recursiva aprovechamos la *pila de ejecución* en lugar de crear y manejar una pila por nuestra cuenta.

Funciones Anónimas

Las funciones pueden ejecutarse de forma **anónima**, o sea que no necesitan un nombre de variable para ser ejecutadas.

¿Recuerdas que la *expresión de llamado* consiste de una *expresión* seguida inmediatamente de paréntesis? Volviendo al ejemplo de `sumar`, podríamos declarar y ejecutar la Función en *el lugar* de la siguiente manera:

```
ENVIAR (Función(a, b)
    DEVOLVER a + b
FIN)(2, 2) //Envía "4"
```

Aquí, estamos creando una Función con una expresión literal encerrada entre paréntesis y ejecutándola inmediatamente después, abriendo otros paréntesis donde pasamos los argumentos separados por comas.

Nótese que la Función expresada no está asociada a ninguna variable, o sea que **va a ser olvidada después de ejecutarse**. Esta expresión literal de Función se evalúa como un valor de Función y se lo ejecuta inmediatamente cuando se vuelve operando de una expresión de llamado.

Funciones como Parámetros

Como las Funciones son **valores**, también pueden ser **argumentos** de otra Función.

En el siguiente ejemplo, `cambiarLista` es una Función que debe tomar una Lista `li` y una Función `fn`. Se ejecuta `fn` para cada elemento de `li`:

```
CARGAR cambiarLista con Función(li, fn)
PARA i desde 0 hasta li→largo
    CARGAR li→(i) con fn(li→(i))
FIN
FIN
```

Con esto en mente, vamos a crear una variable `duplicar` que representa una Función que devuelve el doble de lo que ingreses. Luego, pasémosla a `cambiarLista` en conjunto con una Lista de números (`unaLista`). Esto duplicará el valor de los elementos de `unaLista`:

```
//La Función que diseñamos en la anterior página...
CARGAR cambiarLista con Función(li, fn)
    PARA i desde 0 hasta (li→largo - 1)
        CARGAR li→(i) con fn(li→(i))
    FIN
FIN

//La Función que le pasaremos a la de arriba...
CARGAR duplicar con Función(val)
    DEVOLVER val * 2
FIN

//;Aplicar lo que aprendimos!
CARGAR unaLista con Lista 6, 2, 4, 7, 10
EJECUTAR cambiarLista(unaLista, duplicar)
ENVIAR unaLista→unir(", ") //Envía "12, 4, 8, 14, 20"
```

Y bien podríamos usar una Función anónima en lugar de guardar una variable de Función solo para esta operación:

```
//La Función que diseñamos en la anterior página...
CARGAR cambiarLista con Función(li, fn)
    PARA i desde 0 hasta (li→largo - 1)
        CARGAR li→(i) con fn(li→(i))
    FIN
FIN

//;Aplicar lo que aprendimos!
CARGAR unaLista con Lista 6, 2, 4, 7, 10

EJECUTAR cambiarLista(unaLista, Función(val) DEVOLVER val * 2 FIN)

ENVIAR unaLista→unir(", ") //Envía "12, 4, 8, 14, 20"
```

Ejemplo para Más Tarde

Ahora que entendemos mejor las implicaciones de que las funciones se traten como cualquier otro valor, considera el siguiente código:

```
//Esta sentencia es nueva, no te preocupes si no entiendes
LEER Texto acción con "gritar"

CARGAR hacerAlgo con Nada

SI acción es "gritar"
    CARGAR hacerAlgo con Función()
    ENVIAR "AAAAAAAAAAAAAAAAAAAAAAA"
    FIN
SINO SI acción es "correr"
    CARGAR hacerAlgo con Función()
    ENVIAR "Trotando!!!!!!"
    FIN
SINO
    CARGAR hacerAlgo con Función()
    ENVIAR "¿...no se te ofrece nada?"
    FIN
FIN

//Realiza una acción diferente dependiendo del valor de "acción"
EJECUTAR hacerAlgo()
```

Aquí, se ejecuta una función `hacerAlgo` **diferente** dependiendo de la acción ingresada por el Usuario (lo explicaremos luego).

Ámbito de Función

Las variables declaradas dentro de una Función **no** pueden accederse fuera de esta. Sin embargo, una Función *puede* acceder tanto a las variables del ámbito en el que se la **declara** como a las del ámbito en el que se la **llama**.

```
CARGAR a con 1
CARGAR b con 2
CARGAR c con 30

CARGAR sumarInterno con Función()
    CARGAR a con 5
    CARGAR b con 50
    CARGAR c con 500
    DEVOLVER a + b + c
FIN

ENVIAR sumarInterno() //Envía "555" (5 + 50 + 500)

CARGAR sumarExternoDeclaración con Función()
    DEVOLVER a + b
FIN

ENVIAR sumarExternoDeclaración() //Envía "3" (1 + 2)

CARGAR sumarExternoLlamado con Función()
    CARGAR a con 10
    CARGAR b con 20
    CARGAR d con 30
    DEVOLVER a + b + c + d
FIN

CARGAR d con 400
ENVIAR sumarExternoLlamado() //Envía "460" (10 + 20 + 30 + 400)
```

Por este motivo y otros, de nuevo, ten cuidado al modificar una Lista o un Registro externo a una Función.

Funciones Anidadas

Puedes crear varias funciones anidadas, y funciona como te esperarías:

Como vemos, la Función `vivir` declara las funciones `comer` y `dormir` *al ejecutarla*. Esto significa que estas 2 funciones no están declaradas fuera de `vivir`.

Funciones como Retorno

Como las funciones son **valores**, pueden ser asignadas a otras variables mediante su identificador o devueltas de una Función de igual forma:

```
CARGAR fac con factorial //Reasignación  
DEVOLVER factorial      //Retorno
```

Veamos ambos casos en este ejemplo:

```
CARGAR unaFunción con Función()
    CARGAR otraFunción con Función()
        ENVIAR "Esternocleidomastoideo"
    FIN
    DEVOLVER otraFunción
FIN
CARGAR fn con unaFunción()
EJECUTAR fn() //Envía "Esternocleidomastoideo"
```

Múltiples Funciones Anidadas

Retomando el ejemplo anterior: puedes declarar múltiples funciones anidadas, lo cuál hace que tengan el ámbito de la Función que las contiene, de forma recursiva.

Hagamos un ejemplo:

- Una Función `a` contiene una Función `b`, que contiene una Función `c`.
- La Función `b` tiene acceso a la Función `a`, y la función `c` tiene acceso al a función `b`.
- Como `b` puede acceder todo lo de `a`, `c` también puede.

```
CARGAR a con Función(v)
    CARGAR b con Función(w)
        CARGAR c con Función(x)
            ENVIAR v + w + x
        FIN
        EJECUTAR c(3)
    FIN
    EJECUTAR b(2)
FIN

EJECUTAR a(1) //Envía "6" (1 + 2 + 3)
```

En este ejemplo, `c` accede `b`, y por medio de `b`, accede `a`.

Nótese que acceder `c` desde `a` no es posible, porque la recursividad de ámbitos no se aplica al revés. `c` solo es accesible por `b`.

Mismo Identificador en Diferentes Funciones

Así como es posible tener múltiples variables con el mismo nombre pero declaradas en diferentes bloques, es posible algo similar con funciones.

Nuevamente, al igual que con los bloques, las variables declaradas en un punto más anidado tienen prioridad sobre el resto:

```
CARGAR var con "caramelo"
CARGAR fn con Función(var)
    ENVIAR var
FIN

EJECUTAR fn("chocolate") //Esto envía "chocolate"

ENVIAR var //Esto envía "caramelo"
```

Con esto deberías tener una buena idea de cómo funcionan las Funciones. Úsalas siempre que lo veas conveniente. Permite que tu código sea más fácil de leer, más fácil de modificar o mantener y evita reutilizar tanto código.

A continuación, se dará un pequeño paseo por algunas **Funciones nativas**.

Funciones Nativas

Las **funciones nativas** son *siempre* accesibles en cualquier *Tubérculo*. Están incrustadas directamente en el *código fuente de PuréScript*, así que son más rápidas de ejecutar y en algunos casos ofrecen capacidades que no serían posibles con las otras herramientas disponibles en el lenguaje.

Las funciones nativas *pueden* acceder al ámbito externo, pero el ámbito externo **no** puede acceder al ámbito de las nativas. Algunas funciones nativas ofrecen interoperatividad adicional con Discord, uno de los motivos por los cuales son tan útiles.

Por ejemplo, estas son funciones nativas para verificación de Tipo de valor:

A tener en cuenta...

Este es un listado reducido de las funciones nativas de PuréScript. Existen más funciones nativas designadas para otras tareas. Puedes ver el listado completo [aquí](#).

`esNada(expresión)`

Comprueba si una `expresión` evalúa al valor especial `Nada`.

`esNúmero(expresión)`

Comprueba si una `expresión` es de tipo `Número`.

`esTexto(expresión)`

Comprueba si una `expresión` es de tipo `Texto`.

`esLógico(expresión)`

Comprueba si una `expresión` es de tipo `Lógico`.

`esLista(expresión)`

Comprueba si una `expresión` es de tipo `Lista`.

`esRegistro(expresión)`

Comprueba si una `expresión` es de tipo `Registro`.

Métodos

Un método es una **Función** que es **miembro** de un valor. Los métodos, al igual que los elementos de Lista o las entradas de Registro, se acceden mediante el operador de flecha.

Como ya vimos repetidas veces antes, solo que sin explicarlo del todo, las Listas tienen el método `unir`. `unir` básicamente devuelve un Texto con todos los elementos de la Lista, unidos con el Texto indicado de argumento entre cada uno. En este caso, ", ".

```
CARGAR primates con Lista "Mono", "Orangután", "Gorila", "Lémur"  
//Envía "Mono, Orangután, Gorila, Lémur"  
ENVIAR primates→unir(", ")
```

Algo particular de los métodos es que los tipos de Número y Texto también tienen algunos métodos por su cuenta. Si bien habíamos mencionado que los valores de dichos tipos no tienen miembros accesibles, los *tipos* en sí de los mismos ofrecen funcionalidad adicional a través de métodos.

Estos métodos son accedidos con el mismo operador de flecha. Puedes revisarlos todos más tarde en la [Referencia](#), de momento daremos un paseo general por algunos de los **métodos nativos** más importantes.

Los *tipos* ofrecen métodos que son accesibles por cualquier valor de dicho tipo. Hay algunos tipos que no tienen métodos de tipo: Lógico, Función y Nada.

A tener en cuenta...

Se dará un listado reducido de métodos nativos. Ten en cuenta que existen más métodos de tipo y que no se entra en tantos detalles en esta sección. Puedes ver el listado completo y detallado en la [Referencia](#).

Ejemplos de Métodos de Número

Un método sencillo de entender es `Número→aTexto()`:

```
<número>→aTexto()
```

Es funcionalmente idéntico a una conversión a Texto con la expresión `Texto <número>`. Devuelve una representación textual del Número al que pertenece el método.

```
3.1415→aTexto() //Evalúa a un Texto "3.1415"
```

Nótese que `Número→aTexto()` **no modifica** el valor original, sino que *devuelve* un Texto.

Otros métodos sencillos pero con mayor utilidad son `Número→suelo()`, `Número→techo()` y `Número→redondear()`:

```
<número>→suelo()  
<número>→techo()  
<número>→redondear()
```

Estos 3 aplican diferentes operaciones de redondeo al valor. `Número→suelo()` redondea hacia abajo, `Número→techo()` redondea hacia arriba y `Número→redondear()` redondea al entero más cercano.

```
3.1415→suelo() //Evalúa al Número 3  
2.7182→suelo() //Evalúa al Número 2  
  
3.1415→techo() //Evalúa al Número 4  
2.7182→techo() //Evalúa al Número 3  
  
3.1415→redondear() //Evalúa al Número 3  
2.7182→redondear() //Evalúa al Número 3
```

Los métodos de redondeo **no modifican** el valor original, sino que *devuelven* otro Número.

Por último, otro método que está bueno conocer es Número→formatear():

```
<número>→formatear(acortar, mínimoDígitos)
```

Es similar a Número→aTexto() pero ofrece opciones adicionales de formato.

```
3.1415→formatear(Falso, 1) //Evalúa a un Texto "3.1415"  
3.1415→formatear(Falso, 3) //Evalúa a un Texto "003.1415"  
123456789.5→formatear(Falso, 1) //Evalúa a un Texto "123,456,789.5"  
123456789.5→formatear(Verdadero, 1) //Evalúa a un Texto "123.456 millones"  
1234567.5→formatear(Verdadero, 3) //Evalúa a un Texto "001.234 millones"
```

Ejemplos de Métodos de Texto

Comencemos con Texto→aLista():

```
<texto>→aLista()
```

Devuelve una representación de Lista cuyos elementos son cada carácter del Texto.

Contrario a Número→aTexto(), no es posible convertir un Texto a una Lista sin este método.

```
"Mono"→aTexto() //Evalúa a una Lista de elementos: "M", "o", "n", "o"
```

Texto→aLista() **no modifica** el valor original, sino que *devuelve* una Lista.

Existen los métodos Texto→aMinúsculas() y Texto→aMayúsculas():

```
<texto>→aMinúsculas()  
<texto>→aMayúsculas()
```

Hacen lo que te esperarías.

```
"Lémur"→aMinúsculas() //Evalúa a un Texto "Lémur"  
"Lémur"→aMayúsculas() //Evalúa a un Texto "LÉMUR"
```

También existen varios métodos de comprobación para Textos:

```
<texto>→comienzaCon(caracteres)  
<texto>→terminaCon(caracteres)  
<texto>→contiene(caracteres)
```

Comprueban si un Texto comienza con, termina con o contiene, respectivamente, una secuencia de caracteres.

```
"Orangután"→comienzaCon("Oran") //Evalúa a Verdadero  
"Orangután"→comienzaCon("Lé") //Evalúa a Falso  
  
"Gorila"→terminaCon("la") //Evalúa a Verdadero  
"Gorila"→terminaCon("LA") //Evalúa a Falso  
  
"Mono"→contiene("Mono") //Evalúa a Verdadero  
"Mono"→contiene("a") //Evalúa a Falso
```

Los Textos son de los que más métodos de tipo tienen, así que asegúrate de echarles un ojo en la [Referencia](#) después.

Ejemplos de Métodos de Lista

Veremos más métodos de Lista más adelante. De momento, veamos algunos sencillos. Los valores de tipo Lista son los primeros que cuentan con métodos que **modifican** su valor en lugar de solo devolverlo. No te preocupes, se indicará claramente cuando sea el caso.

Comencemos con `Lista→aRegistro()`:

```
<lista>→aRegistro()
```

Devuelve un Registro cuyas entradas corresponden a los elementos de la Lista.

```
(Lista "A", "B", "C")→aRegistro() //Evalúa: Registro 0: "A", 1: "B" 2: "C"
```

Las claves de la Lista generada corresponden al índice del elemento original. El valor de la entrada es el elemento en cuestión.

Sigamos con `Lista→contiene()`:

```
<lista>→contiene(valor)
```

Similar a `Texto→contiene()`, indica si existe un valor en la Lista que equivalga al indicado.

```
(Lista 1, 2, 3)→contiene(2) //Evalúa a Verdadero  
(Lista 4, 5, 6)→contiene(Verdadero) //Evalúa a Falso
```

Sigamos con `Lista→aInvertida()` y `Lista→invertir()`:

```
<lista>→aInvertida()  
<lista>→invertir()
```

`Lista→aInvertida()` devuelve una Lista con los mismos elementos que la original pero ordenados al revés. `Lista→invertir()` en cambio invierte los elementos **de la original**.

```
(Lista 1, 2, 3)→aInvertida() //Evalúa a: Lista 3, 2, 1  
(Lista 4, 5, 6)→invertir() //Evalúa a Nada, pero modifica La Lista
```

Nótese que, incluso si `Lista→aInvertida()` devuelve una Lista distinta, los elementos de la Lista en sí son los mismos, no una copia de los originales. Por ende, si un elemento de la otra Lista fuese a ser modificado, se vería reflejado en la Lista original y vice-versa.

Los siguientes métodos de Lista que veremos más adelante van a ser mucho más expresivos a costa de ser más complejos.

Ejemplos de Métodos de Registro

Ten en cuenta que los métodos de Registros no pueden ser sobre-escritos. Esto significa que incluso si usas CARGAR para asignar una clave que se llame igual que uno, no servirá.

```
CARGAR gnomos con Registro rojo: "Traybar", verde: "Grawin"
ENVIAR gnomos→contiene //Envía "[Función nativa]"

CARGAR gnomos→contiene con 42 //Esto NO alza un error, ¡cuidado!
ENVIAR gnomos→contiene //Envía "[Función nativa]" de igual manera
```

Lo mismo aplica para las entradas Registro→largo o Registro→tamaño (equivalentes).

Entre los métodos de Registro, podemos destacar Registro→claves, Registro→valores y Registro→entradas:

```
<registro>→claves()
<registro>→valores()
<registro>→entradas()
```

Estos métodos devuelven Listas con las claves, los valores y las entradas, respectivamente, del Registro que llama el método.

```
(Registro a: "Z", b: 42)→claves() //Evalúa: Lista "a", "b"
(Registro a: "Z", b: 42)→valores() //Evalúa: Lista "Z", 42

//Se evalúa como una Lista de Listas. La Lista interna es la entrada:
//Lista (Lista "a", "Z"), (Lista "b", 42)
(Registro a: "Z", b: 42)→entradas()
```

¡Advertencia!

Los Registros no son un conjunto *ordenado*. Por ende, el orden de los elementos al convertir un Registro a una Lista **no está asegurado**.

Veremos los métodos de Marco más adelante. Antes de continuar con métodos más avanzados, veamos otro tema relativamente importante a continuación...

Funciones Lambda

Previamente vimos que podemos pasar funciones como parámetros a otra Función.

```
cambiarLista(unaLista, Función(val) DEVOLVER val * 2 FIN)
```

Existe un tipo de *expresión de Función* diseñado para simplificar operaciones sencillas como esta. Las **expresiones de Función Lambda** usan la siguiente sintaxis para definir una Función simple que toma N parámetros y devuelve la evaluación de una expresión:

```
(<arg1>, <arg2>, (...), <argN>) ⇒ <expresión>
```

Con esto, el anterior código queda simplificado así:

```
cambiarLista(unaLista, (val) ⇒ val * 2)
```

Lo que define a la expresión Lambda es el operador de flecha gorda (\Rightarrow). Los paréntesis son *opcionales* si hay **exactamente** 1 argumento.

Expresiones de Secuencia

Una **expresión de secuencia** consiste de múltiples expresiones separadas por comas y encerradas entre paréntesis. Al evaluarse, se evalúan *todas las expresiones contenidas en el orden que fueron escritas*.

El resultado de una expresión de secuencia siempre es el valor evaluado de la **última** sub-expresión, por lo que las expresiones previas son irrelevantes sin efectos secundarios.

```
CARGAR li con Lista "a", "b", "c"  
CARGAR agregarUno con Función(li, x) EXTENDER li con x FIN  
//Envía "a, b, Z"  
ENVIAR (li→robarÚltimo(), agregarUno(li, "Z"), li→unir(", "))
```

(Nota: *Lista→robarÚltimo()* remueve el último elemento de la Lista y lo devuelve).

Métodos Nativos de Orden Superior

Una Función de Orden Superior es la manera formal de decirle a aquellas Funciones que toman una o más **funciones como argumentos**. Esta sección va a cubrir varios métodos importantes de entre estos.

Métodos de Lista avanzados

Las Listas tienen varios métodos de orden superior. Empecemos con `Lista→paraCada()`:

```
<lista>→paraCada(procedimiento)
<lista>→paraCada((elemento, índice, listaOriginal) ⇒ <expresión>)
```

Este método tiene un comportamiento similar a utilizar `PARA CADA` para iterar sobre cada elemento de una Lista, con algunas utilidades adicionales. El método como tal siempre devuelve `Nada`, y la Función de `procedimiento` que pases puede tomar hasta 3 argumentos:

1. El valor del **elemento** de la iteración actual
2. El **índice** de la iteración actual
3. El valor de la **Lista** sobre la que se está operando

Como depende de efectos secundarios, en realidad no se lo suele usar con Lambdas.

Veamos ambos casos de todas formas:

```
CARGAR gritar con Función(var)
    ENVIAR var + "!!!"
FIN

CARGAR li con Lista "A", "B", "C"

//Evalúa como Nada, pero hace que se envíe:
//"/0: A!!! " "1: B!!! " "2: C!!! "
EJECUTAR li→paraCada((e, i) ⇒ gritar(i + ":" + e))
```

```
CARGAR li con Lista "A", "B", "C"

EJECUTAR li→paraCada(Función(e, i)
    ENVIAR i + ":" + e + "!!! " //Lo mismo que el ejemplo de arriba
FIN)
```

Un método de Lista en el que se querrían usar Lambdas sería `lista→filtrar()`:

```
<lista>→filtrar(predicado)
<lista>→filtrar((elemento, índice, listaOriginal) ⇒ <expresión>)
```

Este método devuelve una nueva Lista con **solo** aquellos elementos que satisfagan el `predicado` indicado (o sea, aquellos elementos para los cuales la Función `predicado` devuelva un valor que pueda convertirse a `Verdadero`).

CARGAR original con `Lista 42, 24, 99, 50, 38, 79, 80, 8, 77`

CARGAR filtrada con `original→filtrar((e) ⇒ e excede 50)`

ENVIAR `original→unir("-") //Envía "42-24-99-50-38-79-80-8-77"`

ENVIAR `filtrada→unir("-") //Envía "99-79-80-77"`

En otras palabras, discrimina entre aquellos elementos que cumplen y no cumplen una condición dada. Dicha condición sería expresada por el `predicado`.

También existen algunos métodos avanzados que modifican el valor original.

Veamos `Lista→aOrdenada()` y `Lista→ordenar()`:

```
<lista>→aOrdenada(criterio)
<lista>→aOrdenada((a, b) ⇒ <expresión>)

<lista>→ordenar(criterio)
<lista>→ordenar((a, b) ⇒ <expresión>)
```

Al igual que con `Lista→aInvertida()` y `Lista→invertir()`, una devuelve una nueva Lista mientras que la otra devuelve `Nada` y en su lugar modifica la Lista original.

`Lista→aOrdenada()` y `Lista→ordenar()` cambian el orden de los elementos de la Lista en base al `criterio` establecido.

La Función de `criterio` debería tener 2 parámetros:

1. `a` — el valor actual
2. `b` — el valor siguiente

Además, debería devolver un Número positivo, negativo ó 0 dependiendo de la *relación entre ambos elementos* para determinar si intercambiarlos de lugar o no:

- **Número negativo** — `a` precede a `b`. **No** se intercambian las posiciones de `a` y `b`
- **Número cero** — `a` es equivalente a `b`. **No** se intercambian las posiciones de `a` y `b`
- **Número positivo** — `a` excede a `b`. **Se intercambian** las posiciones de `a` y `b`

Los llamados a la Función de `criterio` continúan hasta que todos los elementos satisfacen el orden planteado.

Por ejemplo, para ordenar Números de mayor a menor y vice-versa:

```
CARGAR original con Lista 4, 7, 1, 9, 6, 3, 5, 8, 2, 6
CARGAR ordenada con original→ordenada((a, b) ⇒ a - b)
ENVIAR original→unir(".") //Envía "4.7.1.9.6.3.5.8.2.6"
ENVIAR ordenada→unir(".") //Envía "1.2.3.4.5.6.6.7.8.9"
EJECUTAR ordenada→ordenar((a, b) ⇒ b - a)
ENVIAR ordenada→unir(".") //Envía "9.8.7.6.6.5.4.3.2.1"
```

Por último, veamos `Lista→mapear()`.

```
<lista>→mapear(mapeo)
<lista>→mapear((elemento, índice, listaOriginal) ⇒ <expresión>)
```

Este método devuelve una Lista cuyos elementos corresponden al resultado de la Función de `mapeo` por cada elemento.

La Función de `mapeo` puede devolver cualquier tipo de valor, sin conversiones ni nada.

Una forma fácil de visualizar esto es con una Función de mapeo que duplica el valor de cada elemento de la Lista:

```
CARGAR original con Lista 1, 2, 3  
CARGAR mapeada con original→mapear((e) => e * 2)  
ENVIAR original→unir(", ") //Envía "1, 2, 3"  
ENVIAR mapeada→unir(", ") //Envía "2, 4, 6"
```

Dicho de otra forma, se devuelve una Lista que corresponde cada valor de la Lista original a un valor determinado según la Función de mapeo. O en otras palabras, *mapea* una Lista a otra.

Los Registros igual tienen métodos de orden superior, pero no se cubrirá en esta guía. Por favor, revisa el capítulo de [Referencia](#) para conocer más.

Ya se cubrieron los aspectos primarios del lenguaje. Si entendiste hasta aquí, **¡felicidades!** A continuación, veremos cómo darles vida a tus [Tubérculos](#) (no confundir con jardinería).

Ten en cuenta que las siguientes secciones asumen que ya tienes algo de idea de cómo funciona el lenguaje, así que las explicaciones serán más rápidas y/o técnicas.

Recibiendo Entradas de Usuario

Hasta ahora, no hemos trabajado con **Tubérculos** muy variados. A la hora de programar un **Tubérculo**, sentirás bastante seguido la necesidad de recibir datos de parte del [Usuario](#) para manipular el comportamiento del mismo. O sea, una **Entrada de Usuario**.

Las Entradas de Usuario son datos que el Usuario **dispone** o **ingresa** cuando **ejecuta** un **Tubérculo**, con tal de afectar el curso de su ejecución. Puedes verlo como algo similar a pasarle argumentos a una Función... solo que la Función sería el programa entero.

Las Entradas de Usuario que acepta un **Tubérculo** son preparadas por su programador. Esto incluye su *tipo*, *formato* y *cantidad*.

Por ejemplo, si tenemos un **Tubérculo** “sumar” que suma 2 números que quieras, el Usuario podría escribir “**p!tubérculo sumar 4 2**”. En este caso, 4 y 2 son las Entradas de Usuario que se le pasarán al Tubérculo.

p!tubérculo sumar 4 2

The text "p!tuberculo sumar 4 2" is displayed in a large, bold font. Three vertical arrows point upwards from below the text to specific parts of it. The first arrow points to the prefix "p!", labeled "Comando". The second arrow points to the word "tuberculo", labeled "ID de Tuberculo". The third arrow points to the numbers "4 2", labeled "Entradas".

Todas las ejecuciones de **Tubérculo** siguen este formato.

Una forma de implementar el **Tubérculo** anterior sería la siguiente:

```
LEER Número a con 2
LEER Número b con 2
ENVIAR "Resultado: " + (a + b)
```

(Entenderemos este código más a fondo en las siguientes páginas).

Primera Ejecución y Ejecuciones Subsecuentes

Antes de explicar más a detalle cómo recibir Entradas, hay que tener en cuenta una cosa.

Cuando creas un **Tubérculo**, este realizará una **ejecución de prueba** antes de guardarse. Esta también se llama “**primera ejecución**”, y se comporta de manera algo distinta a las **ejecuciones subsecuentes** (cuando el **Tubérculo** ya se guardó y un **Usuario** lo ejecuta).

En este tema vamos a ver las diferencias de ejecución entre la primera ejecución y el resto.

Sentencia LEER

La Sentencia **LEER** permite interceptar una Entrada de Usuario y recibir su valor. A simple vista, tiene un comportamiento similar a otras sentencias de asignación como **CARGAR**, y ofrece múltiples utilidades que veremos pronto. Esta es la sintaxis básica:

```
LEER <tipo> <identificador>
LEER <tipo> <identificador> con <valor de respaldo>
```

En general, todas las formas de esta sentencia asignan un valor al identificador mencionado. Sin embargo, el comportamiento exacto de cada variante depende de si se está realizando una ejecución de prueba o una ejecución normal:

- **Primera Ejecución** — si se indica un **valor de respaldo** como se ve en la demostración de arriba, se asigna ese mismo valor. Si no, se asigna el *valor por defecto* del tipo designado.
- **Ejecución Ordinaria** — el **valor de respaldo** se *ignora* por completo y **se requiere** que el **Usuario** ingrese un valor para asignar. El **Usuario** está obligado a ingresar un valor que sea **convertible** al **tipo** que se pide. Si no se puede convertir o no se ingresa un valor, el **Tubérculo** no se ejecuta.

En el futuro veremos cómo esperar *Entradas de Usuario opcionales*. De momento, todas las Entradas que designemos serán **obligatorias** para el **Usuario**.

Entonces, sabemos que en **ejecuciones posteriores**, el valor del identificador será asignado según un dato que escriba el Usuario al ejecutar el Tubérculo (se guardará la Entrada de Usuario en el Tubérculo).

Las Entradas son leídas una por una, siendo correspondidas a las partes del código que las requieran **en orden de lectura**.

Volviendo al ejemplo anterior, si guardamos el Tubérculo, veremos que las Entradas que especificamos quedaron registradas.

También veremos que quedan asociadas al identificador que ingresamos.

Los **valores de respaldo** – con 2 – que le asignamos a cada una de las Entradas serán ignorados de ahora en adelante y, en su lugar, serán **ingresados** por cualquier Usuario que ejecute el Tubérculo.

Visor de Tubérculos

TuberID	Autor	Versión
sumar	papitaconpure	PS v1.1

Entradas (variante 1 de 1)

(Número) a : Sin descripción
(Número) b : Sin descripción

PuréScript

```
LEER Número a con 2
LEER Número b con 2
ENVIAR "Resultado: " + (a + b)
```

Nótese que el **tipo** de la Entrada **se conserva**, y es el **único** tipo de valor que puede ingresar el Usuario. De lo contrario, se alza un error de Entrada.

Papita con Puré
plt sumar 10 6

Papita con Puré
Bot de Puré BOT

Resultado: 16

Aquí, como podemos ver, al ingresar los valores 10 y 6, son **convertidos** a Números y luego sumados para enviar el resultado de la suma. Recuerda, **los valores se leen en el orden que los dispone el Usuario**, o sea que el 10 se le asigna a la variable a y el 6 se le asigna a la variable b.

Es importante mencionar que no puedes registrar entradas de *Listas*, *Registros*, *Marcos* ni *Funciones* ya que sería muy complicado e incluso confuso de ingresar para el Usuario. Los tipos de Entrada que puedes registrar son *Números*, *Textos* y *Lógicos*.

Siempre se asegurará que los datos sean ingresados y sean del **tipo** deseado, así que no tienes que preocuparte por que la Entrada no exista o sea de un tipo diferente, solo por su valor.

Ejemplo de Entrada de Texto

Puede que recuerdes que vimos brevemente un LEER Texto en un ejemplo de funciones:

```
//Esta sentencia es nueva, no te preocupes si no entiendes
LEER Texto acción con "gritar"

CARGAR hacerAlgo con Nada

SI acción es "gritar"
    CARGAR hacerAlgo con Función()
        ENVIAR "AAAAAAAAAAAAAAAAAAAAAAA"
    FIN
SINO SI acción es "correr"
    CARGAR hacerAlgo con Función()
        ENVIAR "Trotando!!!!!!"
    FIN
SINO
    CARGAR hacerAlgo con Función()
        ENVIAR "?...no se te ofrece nada?"
    FIN
FIN

//Realiza una acción diferente dependiendo del valor de "acción"
EJECUTAR hacerAlgo()
```

Lo cuál se ejecutaría como lo de la derecha:

Esto no involucra ninguna conversión. El Texto se pasa tal como está y de forma directa al Tubérculo.

Nótese cómo el Usuario no tiene la necesidad de poner comillas dobles ("") si solo se ingresa una palabra. Sin embargo, estas son requeridas si se quiere pasar **más de una palabra** por Entrada. Esto es con tal de delimitar correctamente los Textos que se asignan a cada entrada, en caso de que haya más de una Entrada.

Papita con Puré Today at 3:55 PM
plt pedir gritar

 Papita con Puré plt pedir gritar

Bot de Puré APP Today at 3:55 PM
AAAAAAAAAAAAAAAAAAAAAAA

Papita con Puré Today at 3:55 PM
plt pedir correr

 Papita con Puré plt pedir correr

Bot de Puré APP Today at 3:55 PM
Trotando!!!!!! (edited)

Papita con Puré Today at 3:55 PM
plt pedir cualquier cosa

 Papita con Puré plt pedir cualquier cosa

Bot de Puré APP Today at 3:55 PM
¿...no se te ofrece nada? (edited)

Entradas de Lógico

En el caso de registrar Lógicos, el Usuario debe ingresar cualquier palabra que se pueda interpretar como un **estado Lógico**, como por ejemplo:

- "verdadero / falso"
- "activado / desactivado"
- "prendido / apagado"
- "si / no"
- "1 / 0"
- etc...

Ingresar cualquier cosa no relacionada alzará un error.

No se distinguen minúsculas, mayúsculas ni tildes para esto.

(Escribir con tan poco espacio es muy incómodo, no sé de quién habrá sido esta idea).

The screenshot shows a series of messages between a user and a bot. The user sends commands like 'REGISTRAR Entrada confirmar con Verdadero', 'plt disparar Verdadero', 'plt disparar Falso', and 'plt disparar y gritar'. The bot responds with '¡Bam!', 'Espérate que viene la poli', and an error message about 'TuberInitializerError'.

Papita con Puré plt -cs disparar REGISTRAR Entrada confirmar con Verdadero
Bot de Puré BOT Today at 10:47 AM
¡Bam!

Papita con Puré Today at 10:47 AM
plt disparar Verdadero

Papita con Puré plt disparar Verdadero
Bot de Puré BOT Today at 10:47 AM
¡Bam!

Papita con Puré Today at 10:48 AM
plt disparar Falso

Papita con Puré plt disparar Falso
Bot de Puré BOT Today at 10:48 AM
Espérate que viene la poli

Papita con Puré Today at 10:51 AM
plt disparar y gritar

Papita con Puré plt disparar y gritar
Bot de Puré BOT Today at 10:51 AM

Error de PuréScript
⚠️ TuberInitializerError
Error de inicialización
Se esperaba "Verdadero" o "Falso" en entrada de Tubérculo

Entrada de Lógico en PuréScript v1.0

Entradas de Usuario Opcionales

Ahora explicaremos cómo designar **Entradas opcionales**. Para esto, necesitamos explicar una sintaxis adicional para la Sentencia LEER.

Con la palabra clave `opcional`, conseguimos 4 formas en total de usar LEER:

```
LEER <Tipo> <identificador>
LEER <Tipo> <identificador> con <valor de respaldo>
LEER <Tipo> opcional <identificador>
LEER <Tipo> opcional <identificador> con <valor de respaldo>
```

Y... **¡es así de simple!** Usando la palabra clave `opcional`, permites que el Tubérculo se ejecute incluso cuando el Usuario no provee un dato para una Entrada. Nótese, eso sí, que **todas** las Entradasopcionales van **después** de las Entradas requeridas.

En el caso de que el Usuario no disponga un dato para una Entrada opcional, el valor que se le asignará al identificador será el `valor de respaldo` o, en su ausencia, el *valor por defecto* del tipo designado. Puede que hayas notado que este es *exactamente el mismo comportamiento que se ve en la primera ejecución*.

En otras palabras: la palabra clave `opcional` hace que LEER se comporte de la misma forma que en la primera ejecución en los casos que no se ingresa un dato para la Entrada.

Veamos un ejemplo de esto con el Tubérculo “sumar”:

```
LEER Número opcional a
LEER Número opcional b

ENVIAR "Resultado: " + (a + b)
```

Fíjate que esta vez las Entradas `a` y `b` son opcionales y no se indicó un `valor de respaldo`. Esto significa que en la primera ejecución `a` y `b` valdrán `0`, que es el *valor por defecto* del tipo Número. Así mismo, en **ejecuciones posteriores**, si **no se ingresa** una Entrada, esta valdrá `0`. En cambio, si en ejecuciones posteriores el Usuario ingresa, por ejemplo, “`2 4`”, entonces se enviará “Resultado: `6`”.

Limitando Entradas de Usuario

Puedes usar la **Sentencia de Terminación Condicional**, [PARAR](#), para *abortar la ejecución* de un Tubérculo si los datos que ingresa el Usuario son (arbitrariamente) inválidos:

```
PARAR con <mensaje> si <condición>
```

Si la condición indicada se cumple, se abortará el Tubérculo, enviando solamente el mensaje dado en su lugar. La expresión de mensaje será convertida a Texto si no lo es.

```
LEER Número opcional valor con 42
```

```
PARAR con "¡Debes ingresar un número positivo!" si valor no excede 0
```

```
ENVIAR raíz(valor, 2)
```

En este ejemplo, si el Usuario ingresa un Número negativo ó 0, el Tubérculo se abortará antes de llegar a la sentencia ENVIAR raíz(valor, 2).

Como verás, esta sentencia es útil para limitar de forma fácil y rápida el rango de valores que puede ingresar el Usuario.

Entradas Extensivas

La Sentencia LEER, como cualquier otra sentencia, puede estar dentro de un *ciclo iterativo*. Esto significa que puedes ejecutar el mismo LEER más de una vez.

Cuando se lee el **mismo nombre** de Entrada **más de una vez** en la **misma ejecución** de un Tubérculo, esta Entrada se transforma en una **Entrada Extensiva**.

Las *Entradas Extensivas* son más una diferencia para el Usuario que para el programador. Sin embargo, debes tener en cuenta unas cuantas características que las diferencian de Entradas convencionales. Esas son:

- Un Tubérculo solo puede tener **una** Entrada Extensiva, y debe ser la **última** Entrada. No se pueden tener **dos nombres** extensivos, e intentarlo alzará un error.
- Un nombre de Entrada Extensiva puede recibir una **cantidad indefinida** de valores de parte del Usuario. Los mismos se leen **uno por uno** cada vez que se ejecuta una Sentencia LEER que esté asociada a este nombre.
- Debido a que el nombre puede recibir una **cantidad indefinida** de valores, necesitas una forma de saber *cuándo no quedan más valores para leer*. Para esto, existe la Función nativa `hayEntradas()`, que devuelve Verdadero si quedan Entradas que no han sido interceptadas, o Falso de lo contrario.

¿No tiene mucho sentido? Un ejemplo sencillo debería dejarlo claro:

```
CREAR Lista palabras
PARAR con "Ingresa alguna palabra porfa :(" si no quedanEntradas()
MIENTRAS quedanEntradas()
    LEER Texto palabra
    EXTENDER palabras con palabra
FIN
ENVIAR palabras→mapear(p ⇒ "***" + p + "***")→unir(", ")
```

Esto envía todas las palabras que ingrese el Usuario, en negrita y separadas por comas.

Demostración:

```
p!t -cs listado

CREAR Lista palabras

PARAR con "Ingresa alguna palabra porfa :(" si no quedanEntradas()

MIENTRAS quedanEntradas()
    LEER Texto palabra
    EXTENDER palabras con palabra
FIN

ENVIAR palabras->mapear(p => "**" + p + "**")->unir(", ")
```

— ↶ p!t -cs listado ...
APP Bot de Puré Ingresa alguna palabra porfa :((edited)
p!t listado a b c hola mundo café té
— ↶ p!t listado a b c hola mundo café té
APP Bot de Puré a, b, c, hola, mundo, café, té (edited)

Permitirle al Usuario ingresar una cantidad indeterminada de Entradas sería muy engorroso y *tal vez imposible* de no ser por el mecanismo de Entradas Extensivas.

Es muy importante tener en cuenta este concepto al planificar tus Entradas de Usuario.

Formatos de Entrada

Hay ocasiones en las que puede que no quieras terminar el programa si el usuario ingresa un valor no-tan-válido. También puede que simplemente quieras facilitar y volver más amigable el uso del Tubérculo. LEER tiene una herramienta para esto: **formatos**.

Actualmente, PuréScript cuenta con **dos formatos de Entrada**: uno para Números y otro para Textos. El concepto es muy sencillo de entender y básicamente solo te ahorra escribir.

Algo que sí deberías tener en cuenta, es que los formatos **no** se aplican al **valor de respaldo**, así que asegúrate de que tus valores de respaldo concuerden con el formato que especifiques, a menos que deliberadamente quieras que no sea el caso.

Formato de Rango

Para Entradas de **Número**, puedes indicar un rango *mínimo* y *máximo* de valores. Con este formato, el Número que ingrese el Usuario se mantendrá siempre entre el rango de valores.

La sintaxis hace uso de la palabra clave `entre`, y se escribe así:

```
LEER Número estrellas con 3 entre 1 y 5
ENVIAR "★"→repetido(estrellas)
+ "☆"→repetido(5 - estrellas)
```

Los formatos siempre se escriben al final de la sentencia y se aplican automáticamente sin que el Usuario tenga que cambiar nada.

Fíjate que el mínimo y el máximo están separados por un conector `y`. Este solo te ayuda a leer. Es obligatorio.

El rango es *inclusivo* en ambos extremos, y no le arroja errores ni nada al Usuario.



Formato de Mayúsculas

Para Entradas de **Texto**, puedes indicar si quieres recibir el valor del Usuario con todas las letras en *mayúsculas* o *minúsculas*.

Esta sintaxis usa la palabra clave `en`. Si quieres recibir todo en minúsculas, se hace así:

```
LEER Texto entrada en minúsculas
```

En cambio, para mayúsculas:

```
LEER Texto entrada en mayúsculas
```

Por ejemplo:

```
LEER Texto grito con "MANZANA" en mayúsculas
ENVIAR "!!! + grito + "!!!"
```

Funciona exactamente como te esperarías:

```
p!t gritar a
- p!t gritar a
APP Bot de Puré !!!A!!!
p!t gritar waa
- p!t gritar waa
APP Bot de Puré !!!WAA!!!
p!t gritar waaaaaa
- p!t gritar waaaaaa
APP Bot de Puré !!!WAAAAAA!!!
```

Trabajando con Marcos

Ya vimos casi todo lo básico en PuréScript, pero todavía no hemos indagado en qué son los Marcos ni cómo utilizarlos. Introdujimos el tipo Marco por allá en el inicio de esta guía, pero todavía no lo hemos ni tocado. En esta sección aprenderemos más al respecto.

Por empezar, en caso de que no sepas o no recuerdes a qué nos referimos por “Marco”, es eso que está a la derecha:

Ahora... ¿quieres aprender cómo crear uno?

Quieres hacerlo, ¿no? Se ve cool, ¿verdad? ¡Verdad!?

Obviamente se ve **magnífico**. A continuación, veremos cómo crear este mismo Marco **y te va a gustar**.



Los Marcos representan un tipo de estructura especial. **No tienen forma literal**, así que no se pueden asignar directamente con `CARGAR`. En cambio, `CREAR` les asigna un valor de Marco vacío por defecto, lo cual nos permite modificarlos con **métodos nativos de Marco**:

```
CREAR Marco unMarco
```

```
EJECUTAR unMarco→asignarTítulo("Esto es un Marco")
EJECUTAR unMarco→asignarColor(colorRojo)
```

```
//Buscamos a Bot de Puré por medio de su ID...
CARGAR bot con buscarMiembro("651250669390528561")
EJECUTAR unMarco→asignarImagen(bot→avatar)
```

```
EJECUTAR unMarco→agregarCampo("Mucho gusto", "Encantado de conocerte")
```

```
ENVIAR unMarco
```

(La Función `buscarMiembro` simplemente busca un [miembro de un servidor de Discord](#) y lo devuelve representado en un Registro con un formato específico).

Nota adicional: los Marcos se pasan por referencia a funciones.

Métodos de Marco

↑ No te asustes por la cantidad de métodos que muestra el ejemplo, están todos listados y explicadas en la [Referencia](#). Se explicarán brevemente algunos de ellos ahora.

Las [métodos de Marco](#) son métodos nativos del tipo Marco, así que los puedes acceder desde cualquier Marco. Nótese que todos estos métodos tienen `asignar` y `agregar` en el nombre, ya que describen cómo se relacionan los datos al Marco.

Estas son 3 funciones de Marco bastante comunes:

`marco→asignarTítulo(título)`

Asigna un `título` al `marco`.

`marco→asignarColor(marco, color)`

Asigna un `color` al `marco`. Específicamente a esa línea gruesa de la izquierda. Puedes expresar colores hexadecimales como "#608bf3" o usar alguna de las variables nativas de colores, como la variable `colorRojo` del ejemplo.

`marco→agregarCampo(marco, nombre, valor)`

Añade un campo al `marco`. Siempre se agrega después del resto de campos. Dicho campo lleva un `nombre` (algo como un título para el campo) y un `valor` que va justo debajo del nombre. Según las reglas de Discord, puedes agregar hasta **25** campos en un solo Marco.

Los colores de Marco también están listados y ejemplificados en la [Referencia](#).

Limitaciones de Marcos

Hay unos cuántos límites a considerar al trabajar con Marcos, de parte de Discord:

- La descripción puede tener hasta **256** caracteres.
- El nombre del autor puede tener hasta **256** caracteres.
- Un mensaje puede tener hasta **10** Marcos.
- Un Marco puede tener hasta **25** campos.
- Los nombres de campo están limitados a **256** caracteres y sus valores a **1024**.
- El total combinado de caracteres en todos los Marcos del mensaje no puede exceder los **6000**.

Fuente: [Documentación de la API de Discord](#).

Persistencia de Datos

Los **Tubérculos** son capaces de **guardar y cargar datos entre ejecuciones**. Esto se conoce como *persistencia de datos*. Veremos los mecanismos para lograr esto.

Guardado de Datos

Para guardar datos de la ejecución actual, se usa la Sentencia **GUARDAR**, cuya sintaxis es similar a una sentencia de asignación:

```
GUARDAR <identificador>
GUARDAR <identificador> con <valor>
```

Esta sentencia guarda el **valor** expresado bajo el identificador indicado, de tal forma que se pueda **recuperar** en otra ejecución si se hace referencia al mismo identificador. En otras palabras, permite guardar una variable y hacer que esta persista entre ejecuciones.

Si no se expresa un valor, se guarda el valor de la variable con el identificador dado.

Carga de Datos y Declaración Condicional

Para recuperar el valor de la variable guardada en la próxima ejecución, usaremos una forma que no hemos visto de la sentencia **CARGAR**.

```
CARGAR <identificador>
```

Como verás, no se hace uso de la palabra clave **con**. Esta forma de **CARGAR** permite *declarar condicionalmente* una variable **sin asignarle un valor**, cosa que no se puede ni con **CREAR** ni con la forma de **CARGAR** tradicional.

Esto significa que:

- **Si la variable no existe** — se declara el identificador con el valor **Nada**.
- **Si la variable existe** — *¡no pasa nada!* El identificador ya está declarado así que no hay que hacer nada, y a diferencia del **CARGAR** tradicional, no pisa el valor que tenía.

Esto es perfecto para lo que queremos hacer. Sin embargo, puede que no se entienda del todo por ahora. Veamos un ejemplo usando CARGAR y GUARDAR juntos para que quede claro:

```
CARGAR var  
  
SI var es Nada  
    CARGAR var con 2  
SINO  
    SUMAR var  
FIN  
  
ENVIAR var  
GUARDAR var
```

El anterior código declara condicionalmente una variable var. Si no estaba declarada, le asigna el valor 2. En cambio, si ya estaba declarada, le va a sumar +1 en cada ejecución.

La verdadera fortaleza de este CARGAR es que nos permite asegurarnos de que una variable siempre exista en el **ámbito** que queremos que exista.

Al final podemos ver cómo se guarda el identificador var con la Sentencia GUARDAR para poder ser recibido en la siguiente ejecución con el CARGAR de más arriba.

Esencialmente, este programa actuaría como una especie de “contador” desde 2. A la derecha se puede apreciar este efecto.

Así de sencillo como se ve, esta es la dinámica general para trabajar con variables persistentes.

En la siguiente página, veremos una forma incluso más compacta con la que podemos cargar datos en algunos casos, usando la forma de CARGAR tradicional.

Papita con Puré Today at 4:36 PM
plt -cs contar

CARGAR var

```
SI var es Nada  
    CARGAR var con 2  
SINO  
    SUMAR var  
FIN
```

ENVIAR var
GUARDAR var

Papita con Puré plt -cs contar CAP
Bot de Puré APP Today at 4:36 PM
2 (edited)

Papita con Puré Today at 4:36 PM
plt contar

Papita con Puré plt contar
Bot de Puré APP Today at 4:37 PM
3 (edited)

Papita con Puré Today at 4:37 PM
plt contar

Papita con Puré plt contar
Bot de Puré APP Today at 4:37 PM
4 (edited)

Aprovechando la evaluación de cortocircuito

¿Recuerdas que el operador conector lógico `o` aplica el concepto de *evaluación de cortocircuito*? Bueno, una variable que no está declarada se evalúa como `Nada`, y el valor `Nada` se evalúa como `Falso` al ser convertido a su *representación lógica*, así que podemos sacarle provecho a esto para hacer una especie de “*declaración condicional compacta*” y asignarle un valor a la variable en el caso de que no esté declarada en una sola línea.

Los siguientes dos fragmentos de código son funcionalmente idénticos. Observa:

```
CARGAR var con var o 2
```

```
CARGAR var
```

```
SI var es Nada  
    CARGAR var con 2  
FIN
```

Siempre que quede más claro y sea posible, será preferible usar esta versión compacta. Veamos el ejemplo de la página anterior reescrito de esta forma:

```
CARGAR var con var o 2  
ENVIAR var  
GUARDAR var con var + 1
```

Verás que funciona igual:



Borrando Datos Guardados

Hay veces en las que, en lugar de guardar una variable, queremos evitar que se recupere en la próxima ejecución. Para estos casos, puede que intentemos expresar:

```
GUARDAR var con Nada
```

Pero eso **alzaría un error** porque el tipo Nada es **inválido** para operaciones de guardado.

En cambio, puedes usar la Sentencia [BORRAR](#), que está pensada para este mismo caso:

```
BORRAR var
```

La Sentencia BORRAR simplemente borra cualquier valor que haya sido guardado para ejecuciones posteriores bajo el identificador indicado. **No borra ninguna variable de la ejecución actual que tenga ese identificador.**

Limitaciones de Datos Guardados

Actualmente, PuréScript impone estas limitaciones de guardado de variables:

- Volver a crear un **Tubérculo**, incluso si es bajo el mismo *TuberID*, hace que se olviden todos los datos guardados asociados al mismo. En otras palabras, se perderán todos los datos guardados en un **Tubérculo** al recrearlo.
- El conjunto total de datos guardados no puede exceder los **128KiB** en memoria. Puede parecer poco, pero es más que suficiente para la gran mayoría de programas que se te ocurran en PuréScript.

¡Este es el final de la Guía por ahora!

A continuación, podrías intentar programar tus propios [Tubérculos](#) consultando la [Referencia](#) cada tanto.

También puedes consultar y recrear algunos [Programas de Ejemplo](#).

¡Suerte!

Ilustración por [Rakkidei](#).



Práctica

Este capítulo es una recopilación de programas que valdría la pena tener en un servidor, en lugar de ejemplos aislados como en la [Guía](#) o [Referencia](#). A modo de demostración.

Se recomienda **fuertemente** haber leído una buena porción de la [Guía](#) para tener una noción general del lenguaje antes de siquiera revisar los programas del capítulo.

Para cada programa, se comenzará con una idea básica de lo que se quiere lograr, se dará un código inicial y se lo irá refinando de a poco, explicando en el proceso.

Puedes leer todo completo antes de programar, pero se te incentiva a solo leer el planteo del problema e intentar resolverlo por tu cuenta antes de leer la solución. Cada problema tiene desafíos adicionales que también están ordenados por dificultad.

Los programas a continuación están ordenados de más sencillos a más avanzados.

Índice de Capítulo

Práctica	96
Fácil - Ruleta de Ideas	97
Intermedio - Cálculo de Daño	102
Intermedio - Discoteca Modificable	109
Avanzado - Tablón de Nombres	117
Experto - Batalla Purémon	126

Fácil - Ruleta de Ideas

Situación

¿Son muy indecisos? Bueno, Bot de Puré cuenta con el comando “**p!elegrir**”, pero pueden haber ocasiones en las que tengas una decisión particular muy frecuente en el servidor.

Estaría bueno hacer un **Tubérculo** con opciones predefinidas para resolver esto.

Asumamos que se reúnen frecuentemente en la casa de alguien y usemos estas opciones:

- La casa de Papita
- La casa de Puré
- La casa de Tubérculo
- La casa de Tallo
- La casa de Batata
- La casa de Ramírez “Ratón” Alfonso III

Si no se te ocurre cómo hacer este **Tubérculo**, pasemos al [planeamiento](#).

¿Crees poder hacerlo solo? Adelántate a la [demostración](#) de cómo debería responder el **Tubérculo** y pasemos a los [desafíos](#) cuando hayas terminado de programar.

Planeamiento

Empecemos por lo más fácil y construyamos desde ahí. Este problema en particular es sencillo, pero empezar por lo más fácil **siempre** es buena idea.

Inicialmente sabemos que cada opción será un **Texto**, porque no hay otra forma de representar las opciones que dijimos (tienen letras y tal).

Ahora... ¿cómo guardamos las opciones?

¿Funcionarán variables individuales para cada una?

```
//Opciones
CARGAR v1 con "La casa de Papita"
CARGAR v2 con "La casa de Puré"
CARGAR v3 con "La casa de Tubérculo"
CARGAR v4 con "La casa de Tallo"
CARGAR v5 con "La casa de Batata"
CARGAR v6 con "La casa de Ramírez \"Ratón\" Alfonso III"
```

Bueno... podría funcionar, sí, pero... ¿después cómo hacemos para elegir una al azar?

En realidad, necesitamos que los valores estén relacionados de alguna forma. Para esta tarea en particular, podemos usar la Función nativa [elegir\(\)](#).

Con esto en mente, estamos listos para programar el Tubérculo real.

Programación

¡Escribamos un programa!

En base a lo que planeamos, podríamos tener este código:

```
//Opciones
ENVIAR elegir(
    "La casa de Papita",
    "La casa de Puré",
    "La casa de Tubérculo",
    "La casa de Tallo",
    "La casa de Batata",
    "La casa de Ramírez \"Ratón\" Alfonso III")
```

Y... ¡ya está! Esto hace exactamente lo que queremos. `elegir()` toma una cantidad indeterminada de argumentos y elige uno aleatoriamente.

La desventaja de usar `elegir()`, es que no tenemos control sobre cada uno de los elementos. Podemos darle más flexibilidad al programa usando una Lista.

Elegiremos una opción al azar usando la Función nativa `dado()`, la propiedad `largo` y un acceso computado. Luego, aplicaremos una transformación sencilla al elemento elegido.

```
//Opciones
CARGAR opciones con Lista
    "Papita",
    "Puré",
    "Tubérculo",
    "Tallo",
    "Batata",
    "Ramírez \"Ratón\" Alfonso III",

CARGAR idx con dado(opciones→largo) //Índice aleatorio en "opciones"
CARGAR elegido con opciones→(idx) //Obtener el elemento en ese índice
ENVIAR "La casa de **" + elegido + "**" //Enviar transformado
```

En este ejemplo, nos ahorraremos tener que escribir "La casa de " para cada opción, y de paso hacemos que el nombre aparezca en **negrita** con los ** que lo rodean.

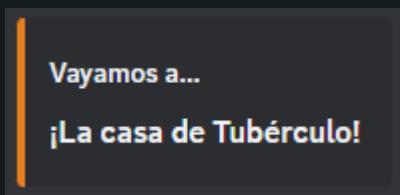
Demostración

Sigue los pasos anteriores y tendrás algo similar a esto:

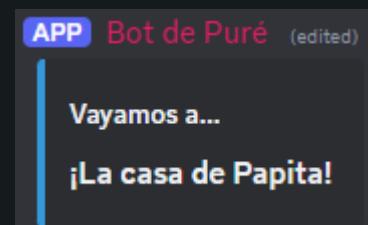
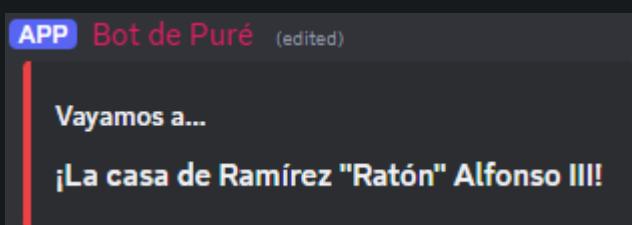
```
p!t reunión
- ↩ p!t reunión
APP Bot de Puré La casa de Puré (edited)
p!t reunión
- ↩ p!t reunión
APP Bot de Puré La casa de Tubérculo (edited)
p!t reunión
- ↩ p!t reunión
APP Bot de Puré La casa de Tallo (edited)
p!t reunión
- ↩ p!t reunión
APP Bot de Puré La casa de Ramírez "Ratón" Alfonso III
p!t reunión
- ↩ p!t reunión
APP Bot de Puré La casa de Tubérculo (edited)
p!t reunión
- ↩ p!t reunión
APP Bot de Puré La casa de Batata (edited)
p!t reunión
- ↩ p!t reunión
APP Bot de Puré La casa de Papita (edited)
```

Desafíos

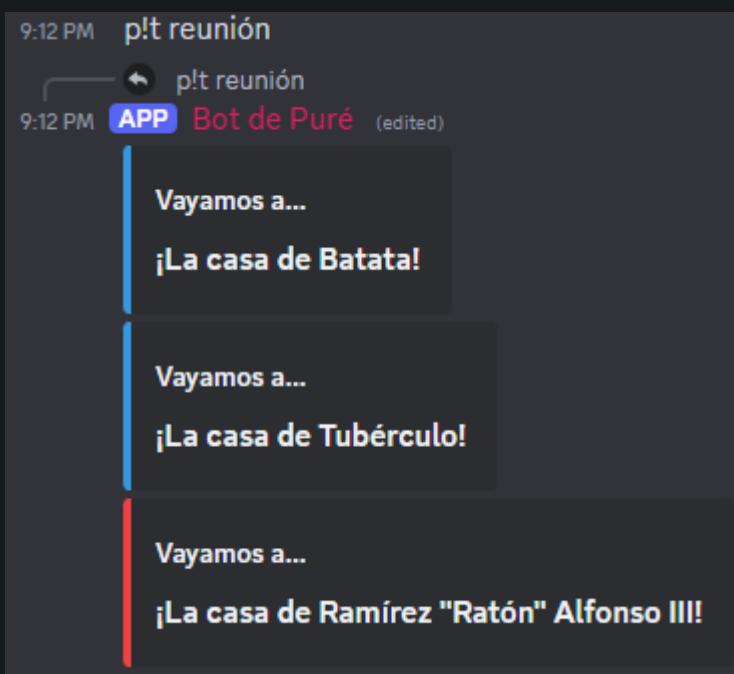
- **Fácil** — Usa un Marco para mejorar la presentación.



- **Normal** — Haz que haya mayor chance de que salga la casa de *Puré*
- **Normal** — Si sale *Ramírez "Ratón" Alfonso III*, ponle un `colorRojo` al Marco. Si sale cualquier otro, ponle un `colorAzul`.



- **Difícil** — Añade la capacidad de enviar 3 opciones al mismo tiempo, sin repetir.
Pista: considera usar el método [Lista→robar\(\)](#).



Intermedio - Cálculo de Daño

Situación

Supongamos que en tu servidor se juega a un juego que tiene un ataque muy fuerte pero que le influyen muchos factores para su efectividad.

Sería genial si se pudiera calcular rápidamente en qué situaciones sirve más...

Es un buen momento para hacer un **Tubérculo** que lo haga todo por ti.

Para mantenerlo sencillo, convengamos que:

- De primeras, el ataque causa **100** de daño si no se considera nada más.
- Si impacta a un duende, hace **+20%** del **daño base**.
- Si impacta a un hada, hace **-50%** del **daño base**.
- Por último: es un ataque mágico, así que los enemigos resistentes a magia reciben **-40%** del **daño total**.

Si no puedes resolver este **Tubérculo**, pasemos al [planeamiento](#) en la próxima página.

¿Crees poder hacerlo solo? Adelántate a la [demostración](#) de cómo debería responder el **Tubérculo** y pasemos a los [desafíos](#) cuando hayas terminado de programar.

Planeamiento

Comencemos súper sencillo para no marearnos. De momento, intentemos calcular el caso en el que se impacta a un **duende hada** que además es **resistente a la magia**... ya sé que suena raro pero no lo cuestiones, es un juego raro.

El objetivo de este programa será calcular e informar el daño final del ataque realizado. Inicialmente, sabemos que siempre tenemos un daño base de **100**:

```
CARGAR daño con 100  
ENVIAR "Daño final: " + daño
```

Además, tenemos que sumar un **20%** de daño por ser un duende y restar un **50%** de daño por ser un hada. Tenemos muchas formas de escribir esto, pero vamos con esta:

```
CARGAR daño con 100  
  
CARGAR dañoDuende con daño * 0.2 //+20%  
CARGAR dañoHada con daño * -0.5 //-50%  
  
//Calculamos el daño total:  
ENVIAR "Daño final: " + (daño + dañoDuende + dañoHada)
```

Por último, tenemos que restar **40%** del daño **total** por la resistencia mágica:

```
CARGAR daño con 100  
  
CARGAR dañoDuende con daño * 0.2 //+20%  
CARGAR dañoHada con daño * -0.5 //-50%  
CARGAR modificación con dañoDuende + dañoHada  
  
CARGAR dañoTotal con daño + modificación //Calculamos el daño total  
CARGAR resistencia con dañoTotal * 0.4 //Restamos 40% al total  
  
ENVIAR "Daño final: " + (dañoTotal - resistencia)
```

Ahora que sabemos cómo calcular uno de los casos, podemos ponernos a programar el Tubérculo final con la mente más clara.

Antes de continuar, analicemos cuáles son los datos que necesitamos recibir de quien ejecute el **Tubérculo**:

1. Requerimos una forma de saber si el ataque **impactó a un duende** o no.
2. Requerimos una forma de saber si el ataque **impactó a un hada** o no.
3. Requerimos una forma de saber si el enemigo impactado **resiste la magia**.

Con esto en mente, inventemos 3 entradas:

Orden	Entrada	Tipo	Descripción
1	esDuende	Lógico	Si impacta a un duende.
2	esHada	Lógico	Si impacta a un hada.
3	esResistente	Lógico	Si impacta a alguien que resiste magia.

Ya estamos listos para programar el **Tubérculo**.

Programación

Una forma de usar las Entradas para calcular el daño sería por medio de condicionales.
Saquémosle provecho a la Sentencia CREAR para esto:

```
LEER Lógico esDuende con Verdadero
LEER Lógico esHada con Verdadero
LEER Lógico esResistente con Verdadero

CARGAR daño con 100
CREAR Número dañoDuende, dañoHada, resistencia //Inicialmente +0%

SI esDuende
    CARGAR dañoDuende con daño * 0.2 //+20%
FIN
SI esHada
    CARGAR dañoHada con daño * -0.5 //-50%
FIN

CARGAR modificación con dañoDuende + dañoHada
CARGAR dañoTotal con daño + modificación

SI esResistente
    CARGAR resistencia con dañoTotal * 0.4 //Restamos 40% al total
FIN

ENVIAR "Daño final: " + (dañoTotal - resistencia)
```

Sin embargo, **esto es mucho texto**. Aprovechemos que los Lógicos se representan numéricamente con 0 para Falso y 1 para Verdadero y simplifiquemos el código:

```
LEER Lógico esDuende con Verdadero
LEER Lógico esHada con Verdadero
LEER Lógico esResistente con Verdadero

CARGAR daño con 100

CARGAR dañoDuende con esDuende * daño * 0.2 //+20% si es duende
CARGAR dañoHada   con esHada   * daño * -0.5 //-50% si es hada

CARGAR modificación con dañoDuende + dañoHada

CARGAR dañoTotal con daño + modificación
CARGAR resistencia con esResistente * dañoTotal * 0.4 //−40% condicional

ENVIAR "Daño final: " + (dañoTotal - resistencia)
```

Como verás, esto simplifica bastante el código.

Lo que estamos haciendo aquí es usar las Entradas mismas como factores para cada modificación que se le aplica al daño.

Por ejemplo, si `esDuende` fuera Falso, al convertirlo a Número daría `0`, lo cual haría que la expresión que se le asigna a `dañoDuende` valga `0`. En cambio, si fuera Verdadero, al convertirlo a Número daría `1`, o sea que no tendría efecto en la operación. Pasa lo mismo con el resto de Entradas.

¡Eso es todo! Si creas el Tubérculo en un servidor, debería funcionar como se espera.

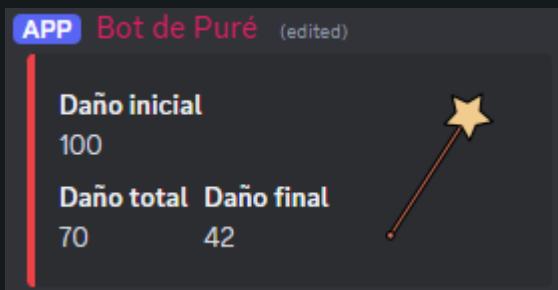
Demostración

Sigue los pasos anteriores y deberías tener algo como esto:

```
p!t cálculo Verdadero Verdadero Verdadero
- ↪ p!t cálculo Verdadero Verdadero Verdadero
APP Bot de Puré Daño final: 42 (edited)
p!t cálculo Falso Falso Falso
- ↪ p!t cálculo Falso Falso Falso
APP Bot de Puré Daño final: 100 (edited)
p!t cálculo si no no
- ↪ p!t cálculo si no no
APP Bot de Puré Daño final: 120 (edited)
p!t cálculo si no si
- ↪ p!t cálculo si no si
APP Bot de Puré Daño final: 72 (edited)
```

Desafíos

- **Fácil** — Haz que se informen tanto el daño inicial como el total y el final.
- **Fácil** — Deja todo bonito usando un Marco para mostrar los datos.



- **Difícil** — Además, haz que se informe el incremento o decremento en base al daño inicial, en forma de porcentaje. Por ejemplo: si el daño final es 100, entonces la relación es de +0%; si el daño final es 72, entonces la relación es de -28%; si el daño final es 120, entonces la relación es de 20%, y así.



Intermedio - Discoteca Modificable

Situación

¿Tus gustos musicales son exorbitantemente buenos y objetivamente los mejores del servidor? ¿Te preguntan todo el rato cuáles son tus temas favoritos?

Obviamente no, pero les vas a refregar tus temas favoritos en la cara igual, ¿no?

Podrías hacer un [Tubérculo](#) para esto.

Realiza un programa que permita incorporar temas a un listado y mostrarlos a gusto.

Por cada tema agregado, se debería:

- Mostrar el **título**
- Mostrar el **autor**
- Mostrar la **portada**
- Dar un **enlace** para escuchar el tema
- **Describir** por qué es el mejor tema de la galaxia

También, los temas deberían estar ordenados según deseas. Para esto podrías darle un puntaje a cada tema.

Además, verifica que los temas solo puedan ser agregados por ti. Uno nunca sabe...

Si no puedes resolver este [Tubérculo](#), pasemos al [planeamiento](#) en la próxima página.

¿Crees poder hacerlo solo? Adelántate a la [demostración](#) de cómo debería responder el [Tubérculo](#) y pasemos a los [desafíos](#) cuando hayas terminado de programar.

Planeamiento

Comencemos por el principio. Necesitamos una Lista de temas y necesitamos guardarla:

```
CARGAR temas con temas o Lista //Cargamos...
```

//Hacemos algo para agregar temas aquí...

```
GUARDAR temas //Guardamos...
```

Además, necesitamos una forma de mostrar un tema individual. Podemos asumir que cada tema es un Registro, pues tienen múltiples propiedades que podemos nombrar. Luego, para mostrarlo simplemente usamos un Marco:

```
//Creamos el Registro del tema...
```

```
CARGAR tema con Registro
```

```
    título: "...con lentitud poderosa",
    autor: "Chris Christodoulou",
    portada: "https://f4.bcbits.com/img/a1339632586\_16.jpg",
    enlace: "https://youtu.be/Nn9trJXUrp0",
    descripción: "Es el mejor tema porque es el mejor tema",
    puntos: 100,
```

```
CREAR Marco m
```

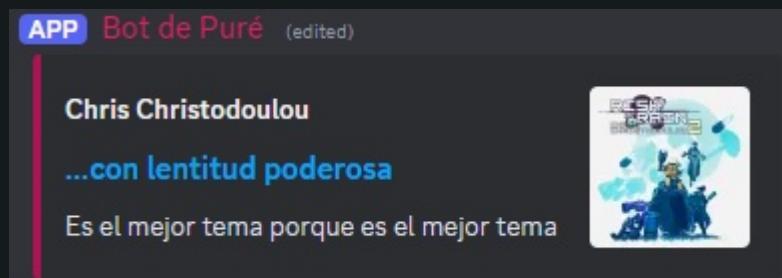
```
//Le asignamos cosas al Marco...
```

```
USAR m→asignarColor(colorRosaOscur)
```

```
    →asignarAutor(tema→autor)
    →asignarTítulo(tema→título)
    →asignarDescripción(tema→descripción)
    →asignarMiniatura(tema→portada)
    →asignarEnlace(tema→enlace)
```

```
ENVIAR m
```

Y más tarde veremos cómo listarlos y ordenarlos todos. De momento, esto está bien.



Con eso fuera del camino, determinemos las Entradas que tendrá el **Tubérculo**. Podemos asumir que todas las Entradas le darán propiedades a un nuevo tema que metamos al listado. Con esto en mente, inventemos las siguientes Entradas:

Orden	Entrada	Tipo	Descripción
1	título	Texto	El título del tema
2	autor	Texto	El autor del tema
3	portada	Texto	El enlace de la portada del tema.
4	enlace	Texto	El enlace para escuchar el tema.
5	descripción	Texto	Por qué crees que el tema es muy bueno.
6	puntos	Número	Puntaje para ordenar el tema.

Puedes ordenar las Entradas como sientas más natural. En esta explicación lo haremos así.

Ya estamos listos para programar el mejor **Tubérculo** del universo concebido.

Programación

Comencemos por terminar de bosquejar la estructura del código que estamos por escribir:

```
//Detener el Tubérculo si estamos en La Ejecución de Prueba
PARAR con "Ejecuta el Tubérculo para agregar temas" si no quedanEntradas()

CARGAR temas con temas o Lista //Cargamos la Lista de temas...

//Leer datos de nuevo álbum
LEER Texto título
LEER Texto autor
LEER Texto portada
LEER Texto enlace
LEER Texto descripción
LEER Número puntos

//Hacemos algo para agregar temas aquí...

GUARDAR temas //Guardamos Los cambios...

//Mostramos el Listado completo de temas aquí...
PARAR con ";No hay temas para mostrar!" si temas→vacía()
```

La verdad, con esto ya casi estamos. Hay 2 cosas por hacer. Atendamos primero el asunto de guardar un Registro del tema nuevo en la Lista. Para esto, vamos a corresponder las Entradas que leímos anteriormente en las respectivas claves de Registro:

```
//Creamos el Registro del tema...
CARGAR tema con Registro
    título:     título,
    autor:      autor,
    portada:   portada,
    enlace:    enlace,
    descripción: descripción,
    puntos:    puntos,

EXTENDER temas con tema //Almacenamos el tema creado en La Lista
```

Lo segundo que debemos atender antes de combinar todas las piezas es ver cómo mostrar todo el listado con cada tema en su propio Marco y ordenar cada tema según sus puntos. Es un poco más fácil de lo que suena. Lo que sí, puede que quieras entender bien cómo usar el Método de Lista [Lista→ordenar\(\)](#). A menos que... ¿quieras ordenar manualmente?

Presta atención. Usaremos el código de hace unas páginas con algunos cambios:

```
//Ordenamos por puntos...
USAR temas->ordenar((t1, t2) => t1->puntos - t2->puntos)

//Mostramos cada tema en su propio Marco...
PARA CADA tema en temas
  CREAR Marco m

  USAR m->asignarColor(colorRosaOscuro)
    ->asignarAutor(tema->autor)
    ->asignarTítulo(tema->título)
    ->asignarDescripción(tema->descripción)
    ->asignarMiniatura(tema->portada)
    ->asignarEnlace(tema->enlace)

  ENVIAR m
FIN
```

¡Y ya está! Ahora podemos mezclar todo lo que vimos en la página que sigue...

```

PARAR con "Ejecuta el Tubérculo para agregar temas" si no quedanEntradas()

CARGAR temas con temas o Lista //Cargamos La Lista de temas...

//Recibimos todos los datos del tema...
LEER Texto título
LEER Texto autor
LEER Texto portada
LEER Texto enlace
LEER Texto descripción
LEER Número puntos

//Creamos y agregamos un nuevo tema al listado...
CARGAR tema con Registro
    título:     título,
    autor:     autor,
    portada:   portada,
    enlace:    enlace,
    descripción: descripción,
    puntos:    puntos,

EXTENDER temas con tema
GUARDAR temas //Guardamos La Lista de temas...

//Ordenamos por puntos...
USAR temas→ordenar((t1, t2) ⇒ t1→puntos - t2→puntos)

//Mostramos el Listado completo de temas después de ordenar...
PARA CADA tema en temas
    CREAR Marco m

    USAR m→asignarColor(colorRosaOscuro)
        →asignarAutor(tema→autor)
        →asignarTítulo(tema→título)
        →asignarDescripción(tema→descripción)
        →asignarMiniatura(tema→portada)
        →asignarEnlace(tema→enlace)

    ENVIAR m
FIN

```

¡Y ya está! Ahora puedes ser más molesto/cool que nunca.

Demostración

Sigue los pasos anteriores y conseguirás un comportamiento similar a esto (hazle zoom):

plt discoteca "...con lentitud poderosa" "Chris Christodoulou" https://f4.bcbits.com/img/a1339632586_16.jpg <https://youtu.be/Nn9trJXUrp0> "Okay en realidad es buen tema pero no realmente el mejor" 88

plt discoteca "...con lentitud poderosa" "Chris Christodoulou" https://f4.bcbits.com/img/a1339632586_16.jpg <https://youtu.be/Nn9trJXUrp0> "Okay en realidad es buen tema pero no realmente el mejor" ...
APP Bot de Puré (edited)

Chris Christodoulou
...con lentitud poderosa
Okay en realidad es buen tema pero no realmente el mejor

plt discoteca "Tenebre Rosso Sangue" "KEYGEN CHURCH" https://f4.bcbits.com/img/a0375793902_16.jpg <https://youtu.be/L5q4uYj-gyg> "No sé si es música católica o música satánica, pero es buena música" 92

plt discoteca "Tenebre Rosso Sangue" "KEYGEN CHURCH" https://f4.bcbits.com/img/a0375793902_16.jpg <https://youtu.be/L5q4uYj-gyg> "No sé si es música católica o música satánica, pero es buena ...
APP Bot de Puré (edited)

KEYGEN CHURCH
Tenebre Rosso Sangue
No sé si es música católica o música satánica, pero es buena música

Chris Christodoulou
...con lentitud poderosa
Okay en realidad es buen tema pero no realmente el mejor

plt discoteca "ピュアヒューリーズ ~ 心の在処" "ZUN" <https://images.genius.com/057ba0d3a6b25e3a45772c56b3adff7b988x988x1.jpg> <https://youtu.be/UcaOpfwgyoM> "De los mejores temas de la saga" 90

plt discoteca "ピュアヒューリーズ ~ 心の在処" "ZUN" <https://images.genius.com/057ba0d3a6b25e3a45772c56b3adff7b988x988x1.jpg> <https://youtu.be/UcaOpfwgyoM> "De los mejores temas de la ...
APP Bot de Puré (edited)

KEYGEN CHURCH
Tenebre Rosso Sangue
No sé si es música católica o música satánica, pero es buena música

ZUN
ピュアヒューリーズ ~ 心の在処
De los mejores temas de la saga

Chris Christodoulou

Desafíos

- **Fácil** — Verifica que el enlace de imagen y tema que se pasen sean válidos... solo para que Bot no te grite cualquier sin-sentido si se pasa un enlace inválido.
(Pista: hay una [Función nativa llamada esEnlace\(\)](#)).
- **Normal** — Puede que no quieras describir por qué todos los temas son buenos. Hay algunos que son objetivamente perfectos, ¿no? Implementa la posibilidad de no ingresar una descripción para algunos temas
(Vas a tener que cambiar el orden de algunas Entradas, eh).
- **Difícil** — Te equivocaste y quieres borrar un tema. Ingéniate una forma de quitar temas del Listado. **¡Cuidado!** Tendrás que lidiar con casos en los que se eliminan todos los temas de la Lista, porque se puede producir un envío vacío.
- **Difícil** — Un momento... ¡no hay forma de simplemente mostrar los temas! Bueno, ya que estamos, agrega una Entrada para decidir si se tiene que "agregar", "quitar" o "mostrar" temas (en lugar de mostrarlos siempre y obligatoriamente agregar o quitar temas). Además, cuando se agrega un tema, se tiene que verificar si el tema ya existe y reemplazarlo si es el caso.
(Pista: considera usar el [método de Lista: Lista→encontrarId\(\)](#), para lo último).

Avanzado - Tablón de Nombres

Situación

Supongamos que quieres mostrar un tablón de nombres de miembros que forman parte de una competencia que se está dando en un servidor.

Los miembros están agrupados en equipos. Todos los equipos tienen un líder, así que un equipo requiere al menos un miembro para existir.

Debes hacer un **Tubérculo** que te permita ingresar equipo por equipo y mostrarlos en Marcos, con el líder y los integrantes del equipo.

El nombre y la foto de perfil del líder deberían mostrarse en el título y la miniatura de su Marco respectivamente. Por cada miembro que no sea el líder, mostraremos su nombre en un campo del Marco.

Cada ejecución del **Tubérculo** debería permitir ingresar o quitar un equipo de la tabla. Para ingresar un equipo, se indican todos los miembros que forman parte del mismo, siendo el primer miembro el líder. Para eliminar un equipo, se indica el líder del mismo.

Si no logras programar este **Tubérculo**, vamos al [planeamiento](#). Si crees poder solo, adelántate a la [demostración](#) y revisa los [desafíos](#) cuando hayas terminado de programar.

Planeamiento

Hagamos un delineado básico de lo que necesitamos para que este programa funcione.

Sabemos que vamos a trabajar con alguna *representación de miembro*. Existe una Función nativa muy útil para obtener un [Registro Estándar de miembro](#), llamada `buscarMiembro()`.

```
CARGAR bot con buscarMiembro("Bot de Puré")
```

También, vamos a trabajar con una Lista que contenga cada uno de estos Registros, porque necesitamos una forma de relacionarlos y ordenarlos. Esto es difícil sin usar Listas.

Además, necesitamos otro Registro que represente un equipo. Este contendrá la Lista de integrantes y discriminará al Líder en otra propiedad.

Por último, necesitamos una Lista que contenga todos los equipos.

Por último, podemos configurar una [Entrada de Usuario Extensiva](#) para recibir miembros uno por uno de parte del usuario. Estas Entradas serán de tipo Texto, pues es el tipo que acepta la Función `buscarMiembro()`:

Orden	Entrada	Tipo	Descripción
1	Líder	Texto	El líder del equipo a agregar o quitar.
2..x	integrantes...	Texto	Cada miembro que no es líder del equipo.

Entonces... tenemos una Lista (equipos) de Registros (equipo) que contienen Listas (integrantes) de Registros (integrantes). ¡Esto se puede complicar!

Con tal de no tropezarnos *tanto*, intentemos primero enviar un solo Marco con un líder e integrantes ya decididos. Un equipo, básicamente.

```
//Esto sería un equipo
CARGAR líder con buscarMiembro("Bot de Puré")
CARGAR integrantes con Lista
    buscarMiembro("Belztyux"),
    buscarMiembro("papitapure"),

//Creamos el Marco del equipo
CREAR Marco m

//Le asignamos las propiedades del Líder y un color cualquiera
USAR m→asignarTítulo("Equipo de " + líder→nombre)
    →asignarMiniatura(líder→avatar)
    →asignarColor(colorDorado)

//Recorremos los integrantes del equipo y los agregamos al Marco (si hay)
SI integrantes→largo
    PARA n desde 1 hasta integrantes→largo
        CARGAR i con integrantes→(n - 1) //Obtenemos el integrante actual
        USAR m→agregarCampo("Integrante " + n, i→nombre, Verdadero)
    FIN
FIN

ENVIAR m
```

Eso quedaría así:



Si más o menos entendiste todo, pasemos programar el Tubérculo.

Programación

Un buen primer paso sería pasar de una vez lo que escribimos antes a un Registro:

```
CARGAR equipo con Registro
    líder: buscarMiembro("Bot de Puré"),
    integrantes: (Lista
        buscarMiembro("Belztyux"),
        buscarMiembro("papitapure"),
    ),
```

Lo segundo sería ver, de forma básica, cómo podemos ir estructurando el código.

```
CARGAR equipos con equipos o Lista //Cargamos...
//Hacemos algo para modificar los equipos aquí...

GUARDAR equipos //Guardamos...

//Recorremos todos los equipos para mostrarlos en Marcos
PARAR con ";No hay equipos para mostrar!" si equipos→vacía()
PARA CADA equipo en equipos
    //Hacemos esto para no andar escribiendo tanto
    CARGAR líder      con equipo→líder
    CARGAR integrantes con equipo→integrantes

    //Esto es el mismo código de la página anterior
    CREAR Marco m

    USAR m→asignarTítulo("Equipo de " + líder→nombre)
        →asignarMiniatura(líder→avatar)
        →asignarColor(colorDorado)

    SI integrantes→largo
        PARA n desde 1 hasta integrantes→largo
            CARGAR i con integrantes→(n - 1)
            USAR m→agregarCampo("Integrante " + n, i→nombre, Verdadero)
        FIN
    FIN

    ENVIAR m
FIN
```

Además, necesitamos un mecanismo para detectar si la primer Entrada se refiere a un líder existente y quitarlo si es el caso.

Para esto, tendremos que buscar el líder dentro de cada equipo:

```
LEER Texto líder
CARGAR líder con buscarMiembro(líder)
PARAR con "Miembro líder no encontrado" si esNada(líder)

CARGAR equipos con equipos o Lista
CARGAR idx con equipos→encontrarId(e ⇒ e→líder→id es líder→id)
SI idx es -1
    //Sin resultados. Aquí vamos a agregar un nuevo equipo...
SINO
    //Hubo un resultado. Aquí vamos a eliminar el equipo encontrado...
FIN

GUARDAR equipos //Guardamos Los cambios...
```

¡Completemos el código de arriba! Veamos cómo tratar el caso de *agregar un equipo*:

```
SI idx es -1
    CREAR Lista listadoIntegrantes

    MIENTRAS quedanEntradas()
        LEER Texto integrantes //Entrada extensiva
        CARGAR miembro con buscarMiembro(integrantes)

        //Verificar que el miembro sea válido y agregarlo a la Lista
        PARAR con "Uno de los integrantes fue inválido" si esNada(miembro)
        EXTENDER listadoIntegrantes con miembro
    FIN

    //Esto lo habíamos programado anteriormente
    CARGAR equipo con Registro
        líder: líder,
        integrantes: listadoIntegrantes,

    EXTENDER equipos con equipo
SINO
```

También debemos tratar el caso en el que se *elimina un equipo*... ah, eso es fácil.

```
SINO
    USAR equipos→robar(idx)
FIN
```

Fíjate cómo tomamos un problema grande y complejo y lo dividimos en partes más simples de entender para resolver todo. ¡Ahora lo que nos queda es conectar las piezas!

```
LEER Texto líder
CARGAR líder con buscarMiembro(líder)
PARAR con "Miembro líder no encontrado" si esNada(líder)

CARGAR equipos con equipos o Lista //Cargamos todos los equipos guardados...
CARGAR idx con equipos→encontrarId(e ⇒ e→líder→id es líder→id)
SI idx es -1
    CREAR Lista listadoIntegrantes

    MIENTRAS quedanEntradas()
        LEER Texto integrantes
        CARGAR miembro con buscarMiembro(integrantes)
        PARAR con "Uno de los integrantes fue inválido" si esNada(miembro)
        EXTENDER listadoIntegrantes con miembro
    FIN

    CARGAR equipo con Registro
        líder: líder,
        integrantes: listadoIntegrantes,

        EXTENDER equipos con equipo
    SINO
        USAR equipos→robar(idx)
    FIN
    GUARDAR equipos //Guardamos los cambios...

//Recorremos todos los equipos para mostrarlos en Marcos
PARAR con "¡No hay equipos para mostrar!" si equipos→vacía()
PARA CADA equipo en equipos
    CARGAR líder con equipo→líder
    CARGAR integrantes con equipo→integrantes

    CREAR Marco m
    USAR m→asignarTítulo("Equipo de " + líder→nombre)
        →asignarMiniatura(líder→avatar)
        →asignarColor(colorDorado)

    SI integrantes→largo
        PARA n desde 1 hasta integrantes→largo
            CARGAR i con integrantes→(n - 1)
            USAR m→agregarCampo("Integrante " + n, i→nombre, Verdadero)
    FIN
    FIN

    ENVIAR m
FIN
```

Puede parecer broma, pero de esto va el proceso de resolución de problemas en programación. *¡Divide y vencerás!* – como dicen por ahí.

Ahora, siempre que hagas programas así de largos en PuréScript, ten en cuenta que tienes un límite de **2000** caracteres por mensaje en Discord. Cosas como acortar nombres de variables pueden servir mucho para no alcanzar este tope.

También recuerda que hay un límite de **1000** créditos implantado en PuréScript, costando **1** cada sentencia y **0.1** cada expresión, así que optimizar puede venir bien.

Dicho esto, **no optimices prematuramente**. El mejor momento para comenzar a preocuparte por optimización es cuando lo veas necesario. Lo importante inicialmente es que el código haga lo que quieras. Luego te preocupas de que lo haga *bien*. Luego, de que lo haga de forma *eficiente*.

Demostración

Este costó, ¿no? Bueno, si hiciste todo bien, deberías poder replicar este resultado final:

p!t tablón papita @Belztyux @Puré

– p!t tablón papita @Belztyux @Puré

APP Bot de Puré (edited)

Equipo de Papita con Puré

Integrante 1 Integrante 2
Belztyux Puré



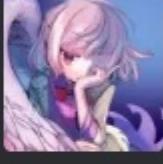
p!t tablón @Bot de Puré @Tubérculo @Batata

– p!t tablón @Bot de Puré @Tubérculo @Batata

APP Bot de Puré (edited)

Equipo de Papita con Puré

Integrante 1 Integrante 2
Belztyux Puré



Equipo de Bot de Puré

Integrante 1 Integrante 2
Tubérculo Batata



p!t tablón @Papita con Puré

– p!t tablón @Papita con Puré

APP Bot de Puré (edited)

Equipo de Bot de Puré

Integrante 1 Integrante 2
Tubérculo Batata



p!t tablón "bot de puré"

– p!t tablón "bot de puré"

APP Bot de Puré ¡No hay equipos para mostrar!

Desafíos

- **Fácil** — Verifica que un equipo no tenga más de 9 integrantes (sin contar al líder).
(Pista: Considera usar la Sentencia PARAR).
- **Normal** — Ordenar los campos de integrantes por nombre de integrante (no líder).
(Pista: Considera usar los [métodos de Lista](#) `Lista→ordenar()` o `Lista→aOrdenada()`).
- **Difícil** — Agrega una Entrada de Texto que permita decidir si "agregar", "quitar" o "mostrar" equipos (en lugar de mostrarlos siempre y agregar o quitar dependiendo de si el líder mencionado existe o no).
- **Difícil** — ¡Es difícil manejar tantas personas! No permitas que se repitan miembros en ningún equipo, considerando tanto líderes como otros integrantes.
(Pista: Considera usar `Lista→encontrar()` o `Lista→encontrarId()`).

Experto - Batalla Purémon

Situación

¡Hagamos un juego!

El juego será una batalla por turnos en la que gana el personaje que quede en pie.

Hacer un Tubérculo de un juego entre 2 usuarios complicaría mucho más un programa que ya de por sí tiene pinta de ser complejo, así que nos limitaremos a hacer un juego que sea de jugador contra máquina. ¡Así es! Será un **Usuario contra Bot de Puré**.

Para no complicarnos tanto, cada personaje tendrá solo 3 propiedades:

- **Nombre** — El primer jugador tendrá el nombre del Usuario, el segundo Bot de Puré.
- **Salud** — Ambos comienzan con 100 de salud.
- **Stamina** — Ambos comienzan con 3. Se recarga 1 por turno si tienen menos de 3.

Además, ambos personajes tendrán estos ataques:

- **Placaje** — hace entre **12** y **32** de daño *inclusive*.
- **Piedrazo** — hace entre **24** y **42** de daño *inclusive*. Cuesta **3 de stamina**.
- **Salpicadura** — técnicamente es un ataque... pero **no hace daño**.

El jugador humano, el Usuario, deberá decidir qué ataque usar en cada Turno. Después del turno del humano, Bot de Puré deberá contraatacar con un ataque al azar.

Al finalizar el turno, se debe mostrar la **salud** y **stamina** de ambos jugadores, o mostrar quién ganó si es que la salud de alguno llegó a 0.

Si se decide un ganador, el próximo uso del Tubérculo debe comenzar una nueva batalla.

Si no resuelves este Tubérculo, vamos al [planeamiento](#). Si realmente crees poder solo, adelántate a la [demostración](#) y revisa los [desafíos](#) luego de programar.

Planeamiento

A este lo vamos a tener que acortar bastante para que entre en un mensaje siquiera...

Lo mejor sería aplicar la de siempre y comenzar por descifrar los problemas más sencillos en lugar de tirarnos de inmediato a resolver todo el Tubérculo. En especial porque este programa demuestra particular complejidad. Es difícil hacer un programa así de la nada.

Esta vez, lo más sencillo va a ser determinar la única Entrada que necesitamos:

Orden	Entrada	Tipo	Descripción
1	ataque	Texto	El líder del equipo a agregar o quitar.

Lo siguiente sería asentar de una vez la estructura de cada personaje. Ambos personajes tendrán la misma estructura, así que de momento solo planteemos uno:

```
CARGAR personaje con Registro  
nombre: "Bot de Puré",  
vida: 100,  
stamina: 100,
```

Ya que los jugadores comparten la misma estructura, no nos vendría mal como siguiente paso determinar qué debemos hacer con los personajes que tratamos:

- Debemos poder crearlos
- Debemos poder hacer que ataquen
- Debemos poder incrementar pasivamente su stamina
- Debemos poder comprobar si ganan

El último paso lo podemos dejar de lado de momento.

Podríamos meter las primeras 3 acciones en funciones, para organizarnos mejor:

```
CARGAR crearPersonaje con Función(nombre)
    DEVOLVER Registro
        nombre: nombre,
        vida: 100,
        stamina: 3,
    FIN

    CARGAR preTurno con Función(pj)
        SI pj->stamina precede 3
            SUMAR pj->stamina
        FIN
    FIN

    CARGAR atacar con Función(pj, ataque, destino)
        //Aquí hacemos que "pj" realice un "ataque" al personaje "destino"
        //Luego vemos cómo hacer esto...
    FIN
```

Esto nos permite tener una forma más intuitiva de manejar cada jugador y visualizar mejor el flujo de nuestro programa y cómo se van modificando los jugadores a medida que el mismo se ejecuta.

Lo último que nos queda es determinar qué hacer con la Entrada `ataque` que inventamos. Podemos deducir que esta estaría involucrada en la Función `atacar` que acabamos de definir. Además, como tenemos un ataque que no hace nada, podemos convenir que si la Entrada no coincide con "`PLACAJE`" o "`PIEDRAZO`", entonces será "`SALPICADURA`".

También, consideremos cómo mostrar los ataques al Usuario. Podríamos hacer una Función separada para simplemente mostrar la cantidad de daño que hizo el ataque, pero esto complicaría las cosas innecesariamente. Mejor simplemente enviar un Marco desde la misma Función `atacar`.

Finalmente, debemos considerar que **Piedrazo** requiere 3 de stamina para usarse, y la consume en el acto.

Como todos los movimientos son ataques, podemos usar la misma plantilla de mensaje para todos. Incluso si **Salpicadura** no hace daño, sigue siendo un ataque.

Esta es la Función `atacar` con la que nos quedaremos. `pj` es el Registro del jugador que **ataca**, mientras que `destino` es el Registro del jugador que **recibe** el ataque:

```
CARGAR atacar con Función(pj, ataque, destino)
CREAR Número daño //Iniciado en 0
SI ataque es "PLACAJE"
    CARGAR daño con dado(12, 33) //12 a 32 de daño inclusive
SINO SI ataque es "PIEDRAZO" y pj→stamina no precede 3
    RESTAR pj→stamina con 3 //Consumir stamina necesaria
    CARGAR daño con dado(24, 43) //24 a 42 de daño inclusive
SINO
    CARGAR ataque con "SALPICADURA" //Por si acaso...
FIN
RESTAR pj→vida con daño

CREAR Marco m
ENVIAR m→asignarColor(colorRojo)→agregarCampo(
    pj→nombre + " usa " + ataque,
    "Causa __" + daño + "__ de daño a **" + destino→nombre + "**")
FIN
```

Lo siguiente sería interceptar la Entrada de Usuario en mayúsculas para determinar el ataque del jugador humano. De paso vemos cómo usar la Función `atacar`:

```
LEER Texto ataque con "PLACAJE" en mayúsculas
USAR atacar(humano, ataque, bot)
```

Ahora lo mismo pero para Bot. Usaremos la Función nativa `elegir()` para su ataque:

```
CARGAR ataque con elegir("PLACAJE", "PIEDRAZO", "SALPICADURA")
USAR atacar(bot, ataque, humano)
```

Lo último *último* antes de programar el código final sería ver cómo determinar si algún jugador ganó. Para esto, verificamos si alguno de los dos personajes murió:

```
SI humano→vida no excede 0 o bot→vida no excede 0
    //...entonces se acabó la batalla...
```

Y con esto ya estamos un poco mejor preparados para programar el Tubérculo final.

Programación

A partir de todo esto, podemos plantear el esqueleto general del programa:

```
//Definimos las funciones que vamos a usar
CARGAR crearPj con nombre => Registro //Expresamos esta en Lambda
    nombre: nombre,
    vida: 100,
    stamina: 3,

CARGAR preTurno con Función(pj)
    SI pj->stamina precede 3
        SUMAR pj->stamina
    FIN
FIN

CARGAR atacar con Función(pj, ataque, destino)
    CREAR Número daño
    SI ataque es "PLACAJE"
        CARGAR daño con dado(12, 33)
    SINO SI ataque es "PIEDRAZO" y pj->stamina no precede 3
        RESTAR pj->stamina con 3
        CARGAR daño con dado(24, 43)
    SINO
        CARGAR ataque con "SALPICADURA"
    FIN
    RESTAR destino->vida con daño

    CREAR Marco m
    ENVIAR m->asignarColor(colorRojo)->agregarCampo(
        pj->nombre + " usa " + ataque,
        "Causa __" + daño + "__ de daño a **" + destino->nombre + "**")
FIN

//Creamos ambos jugadores
CARGAR humano con humano o crearPj(usuario->nombre)
CARGAR bot     con bot     o crearPj("Bot de Puré")

//Recargamos stamina de ambos jugadores
USAR (preTurno(humano), preTurno(bot))

//Realizamos un turno
LEER Texto ataque con "PLACAJE" en mayúsculas
USAR atacar(humano, ataque, bot)

CARGAR ataque con elegir("PLACAJE", "PIEDRAZO", "SALPICADURA")
USAR atacar(bot, ataque, humano)
```

Aunque ese código mida una página entera, lamentablemente todavía le falta un poco para ser el código con el que nos quedaremos.

Comencemos por un problema fácil de arreglar antes de que nos olvidemos. Si el jugador humano mata a Bot, entonces Bot no debería poder atacar porque está muerta:

```
//Realizamos un turno  
LEER Texto ataque con "PLACAJE" en mayúsculas  
USAR atacar(humano, ataque, bot)  
  
SI bot→vida excede 0 //Nos aseguramos de que Bot esté viva para atacar  
    CARGAR ataque con elegir("PLACAJE", "PIEDRAZO", "SALPICADURA")  
    USAR atacar(bot, ataque, humano)  
FIN
```

También, todavía necesitamos una forma de persistir a cada jugador entre ejecuciones. Antes de esto, sin embargo, será mejor que veamos cómo programamos por completo el final de partida, porque nos va a ayudar con el guardado.

Si uno de los dos jugadores pierde, se debe comenzar una nueva batalla. Ahí podemos usar la Sentencia BORRAR, que básicamente borrará los datos del enfrentamiento. En cambio, si ambos personajes siguen en pie, entonces la batalla continúa.

Invirtamos la condición que pensamos antes para comprobar si ambos jugadores siguen en pie. Si no es el caso, borramos sus datos e informamos el final de partida:

```
SI humano→vida excede 0 y bot→vida excede 0  
    //(...Anunciamos la salud y stamina de cada jugador...)  
  
    //Guardar datos de ambos jugadores...  
    GUARDAR humano  
    GUARDAR bot  
SINO  
    //(...Anunciamos el fin de la batalla y el ganador...)  
  
    //Borrar jugadores  
    BORRAR humano  
    BORRAR bot  
FIN
```

Lo único que queda es tener creatividad para los mensajes de final y prepararse mentalmente para ver un código que mide dos páginas (*son menos de 2000 caracteres!*):

```
//Definimos las funciones que vamos a usar
CARGAR crearPj con nombre => Registro
    nombre: nombre,
    vida: 100,
    stamina: 3,

CARGAR preTurno con Función(pj)
    SI pj→stamina precede 3
        SUMAR pj→stamina
    FIN
FIN

CARGAR atacar con Función(pj, ataque, destino)
    CREAR Número daño
    SI ataque es "PLACAJE"
        CARGAR daño con dado(12, 33)
    SINO SI ataque es "PIEDRAZO" y pj→stamina no precede 3
        RESTAR pj→stamina con 3
        CARGAR daño con dado(24, 43)
    SINO
        CARGAR ataque con "SALPICADURA"
    FIN
    RESTAR destino→vida con daño

    CREAR Marco m
    ENVIAR m→asignarColor(colorRojo)→agregarCampo(
        pj→nombre + " usa " + ataque,
        "Causa __" + daño + "__ de daño a **" + destino→nombre + "**")
FIN

//Cargamos ambos jugadores
CARGAR humano con humano o crearPj(usuario→nombre)
CARGAR bot con bot o crearPj("Bot de Puré")

//Recargamos stamina de ambos jugadores
USAR (preTurno(humano), preTurno(bot))

//Realizamos un turno
LEER Texto ataque con "PLACAJE" en mayúsculas
USAR atacar(humano, ataque, bot)

SI bot→vida excede 0
    CARGAR ataque con elegir("PLACAJE", "PIEDRAZO", "SALPICADURA")
    USAR atacar(bot, ataque, humano)
FIN
//(...)
```

```

//(...)

//Mostrar estado
CREAR Marco m
SI humano→vida excede 0 y bot→vida excede 0
  ENVIAR m
    →asignarColor(colorAzul)
    →asignarTítulo("Salud de los contrincantes")
    →agregarCampo(
      humano→nombre,
      humano→vida + " / " + humano→stamina,
      Verdadero)
    →agregarCampo(
      bot→nombre,
      bot→vida + " / " + bot→stamina,
      Verdadero)

//Guardar datos de ambos jugadores...
GUARDAR humano
GUARDAR bot
SINO
  CARGAR ganador con (humano→vida excede 0 y humano→nombre) o bot→nombre
  ENVIAR m
    →asignarColor(colorRojoOscuro)
    →asignarTítulo("¡" + ganador + " ganó!")

//Borrar jugadores
BORRAR humano
BORRAR bot
FIN

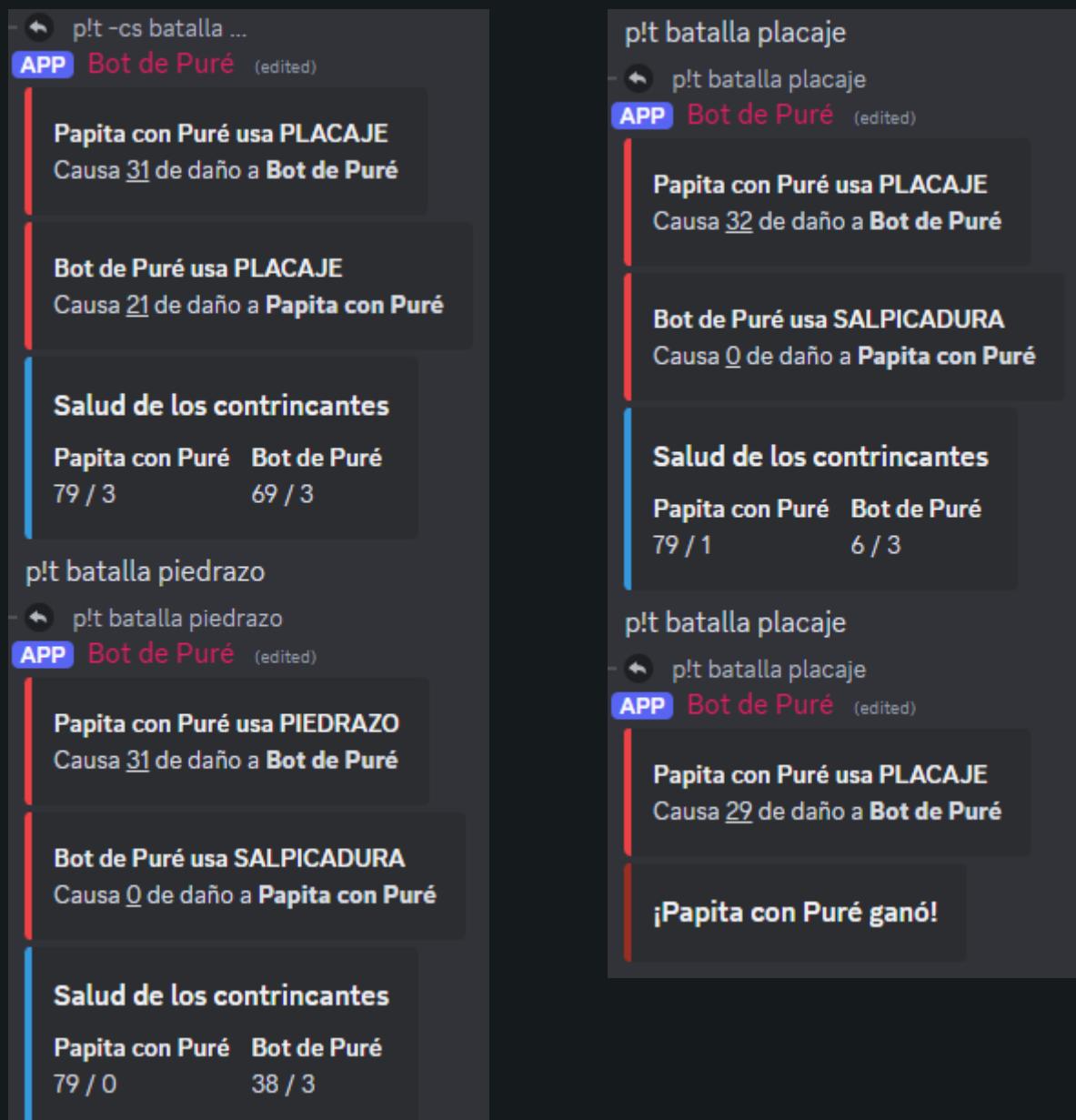
```

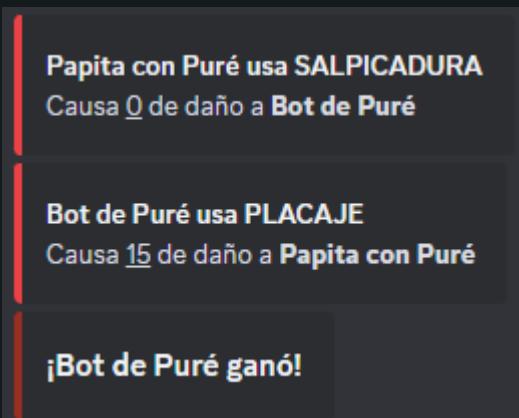
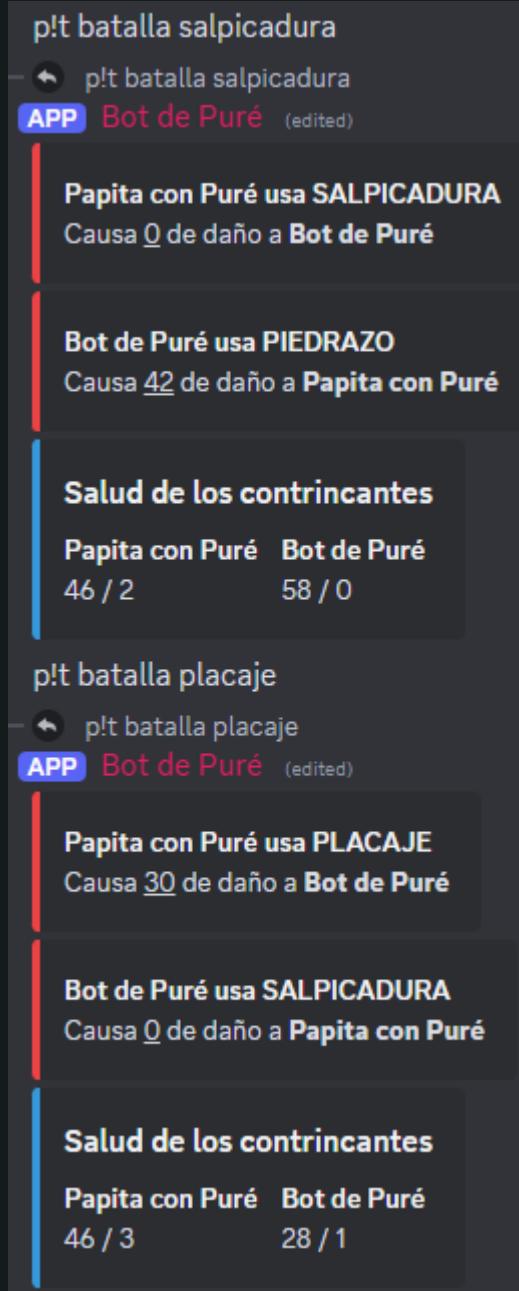
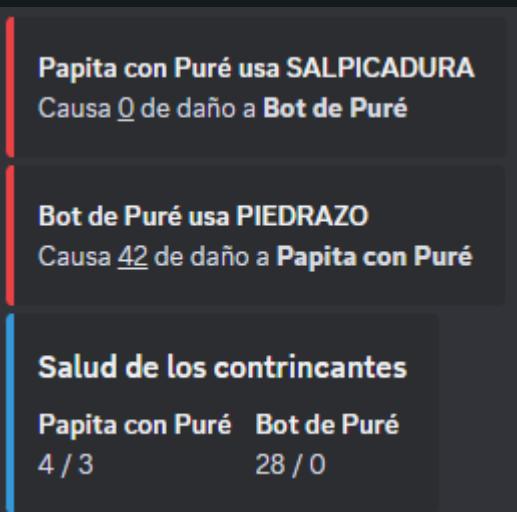
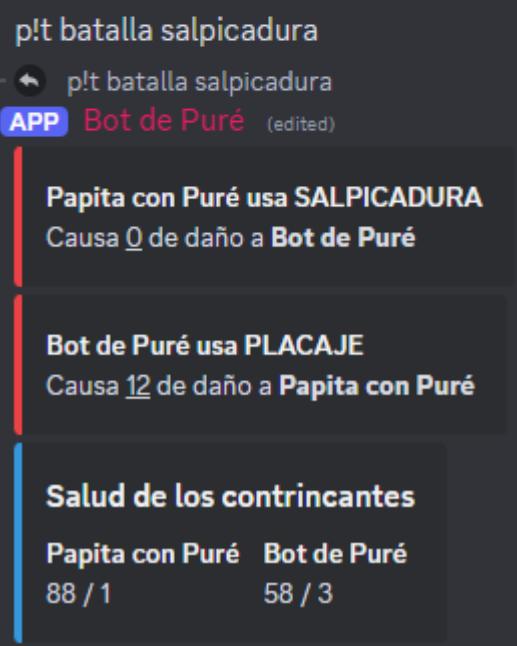
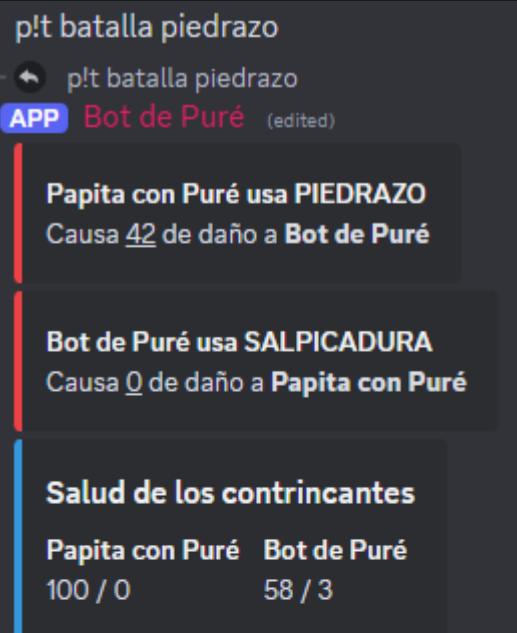
¡Terminamos! Fíjate que por más imposible que parezca un problema, si lo divides en partes y luego las vas relacionando de a poco, resulta mucho más fácil de trabajar.

Nos queda rezar por que no vengan abogados de Puréntendo o PuréFreak a tirarnos abajo el Tubérculo... con lo que nos costó programarlo, sería una verdadera lástima.

Demostración

Si todo salió milagrosamente bien... ¡deberías poder jugar batallas Purémon!

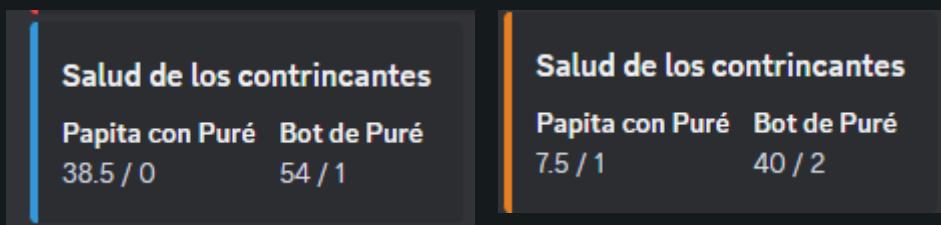




Desafíos

Dado que el programa roza el límite de caracteres sin Discord Nitro, no se puede dar una aproximación muy buena de la dificultad de cada desafío. Se recomienda **no** intentarlos todos al mismo tiempo:

- **Fácil** — Aumenta considerablemente la probabilidad de que Bot use **Placaje**.
- **Normal** — añade probabilidad de **crítico** de **15%** a cualquier ataque. Los ataques críticos hacen el **150% de daño**.
- **Normal** — cambia el color del Marco de estado de la partida a **naranja** cuando la vida del jugador humano caiga por debajo de **30**.



- **Normal** — ¡es demasiado fácil ganarle a Bot! Evita que Bot intente usar **Piedrazo** si **no** tiene suficiente **stamina**. También, haz que siempre lo use si le es posible.

Nótese que con buena **optimización**, un recorte importante a los nombres de variable, sangrías más cortas y algunos sacrificios de **legibilidad**, puede ser posible realizar los **4** desafíos al mismo tiempo. Sin embargo, no te sientas mal si no lo logras, ¡es complicado!

Referencia

Este capítulo cubre todos los indicadores de sentencia, tipos, palabras clave y operadores de PuréScript. Puedes revisar este índice cada vez que tengas una duda sobre algún aspecto del lenguaje. Si no sabes nada del lenguaje, es recomendable leer la [Guía](#).

Índice de Capítulo

Referencia	137
BLOQUE	141
Sintaxis	141
BORRAR	142
Sintaxis	142
Persistencia de Datos de Tubérculos	142
CARGAR	143
Sintaxis	143
Declaración Automática	143
Asignación Condicional	144
Ejemplos	144
COMENTAR	145
Sintaxis	145
CREAR	146
Sintaxis	146
DEVOLVER	147
Sintaxis	147
DIVIDIR	149
Sintaxis	149
EJECUTAR	150
Sintaxis	150
ENVIAR	151
Sintaxis	151
Tipos de Envío	151
Pila de Envíos	152
EXTENDER	153
Sintaxis	153
Indizado	153
FIN	154
Sintaxis	154
Estructuras Compatibles	154
GUARDAR	155



Sintaxis	155
Persistencia de Datos de Tubérculos	155
HACER HASTA	156
Sintaxis	156
LEER	157
Sintaxis	157
Diferencias Situacionales	158
Declaración Automática	158
Directrices de Formato	159
MIENTRAS	160
Sintaxis	160
MULTIPLICAR	161
Sintaxis	161
PARA	162
Sintaxis	162
PARA (Corto)	164
Sintaxis	164
PARA CADA	165
Sintaxis	165
Identificador de Miembro Iterado	165
PARAR	168
Sintaxis	168
REPETIR	170
Sintaxis	170
RESTAR	171
Sintaxis	171
SI	172
Sintaxis	172
SINO	173
Sintaxis	173
SUMAR	174
Sintaxis	174
TERMINAR	176
Sintaxis	176
Número	177
Expresión Literal	177
Valor por Defecto	177
Conversiones de Número	177
Números Negativos	178
Miembros de Número	178
Texto	179
Expresión Literal	179
Valor por Defecto	179

Conversiones de Texto	179
Miembros de Texto	180
Lógico	181
Expresiones Literales	181
Valor por Defecto	181
Conversiones de Lógico	181
Lista	182
Expresión Literal	182
Valor por Defecto	183
Conversiones de Lista	184
Miembros de Lista	184
Registro	185
Expresión Literal	185
Valor por Defecto	186
Conversiones de Registro	186
Miembros de Registro	187
Marco	188
Expresión Literal	188
Valor Inicial	188
Conversiones de Marco	188
Propiedades de Marco	189
Función	190
Expresión literal	190
Valor por defecto	191
Conversiones de Función	191
Métodos	191
Argumentos Opcionales	192
Ámbito de Función	192
Nada	193
Expresión literal	193
Conversiones de Nada	193
Ejemplo	193
Comprobando Nada en una Variable	193
Estructuras Estandarizadas	194
Estructura Estándar "Miembro"	194
Estructura Estándar "Canal"	194
Estructura Estándar "Rol"	194
Estructura Estándar "Servidor"	195
Estructura Estándar "Marco"	195
Variables Nativas	197
Colores	198
Funciones Nativas	200
Utilidad General	201
Comprobación de Tipo	205

Utilidad de Discord	207
Otras Comprobaciones	208
Métodos Nativos	209
Métodos de Número	211
Métodos de Texto	215
Métodos de Lista	220
Métodos de Registro	227
Métodos de Marco	230
Lista de Operadores	234
Operadores Aritméticos	234
Operador de Concatenación	236
Operador de Dos Puntos ":"	236
Operador de Flecha "->"	236
Operador de Acceso Computado "->(...)"	237
Operadores Conectores Lógicos	237
Operador de Negación	238
Operadores Relacionales	238
Operador de Llamado	240
Operador Lambda	240
Operadores de Agrupación "()"	241
Operador de Coma ","	241
Tabla de Precedencias	242

BLOQUE

Indicador de Sentencia » Estructura de Control » Flujo

Sintaxis

BLOQUE

...sentencias a ejecutar dentro del bloque

FIN

El texto oscurecido está a modo de ofrecer contexto y puede que no se escriba exactamente como está.

Inicia un nuevo bloque dentro del actual, creando un nuevo [ámbito de bloque](#) dentro del ámbito actual.

El ámbito del bloque se destruye cuando el bloque termina. Para señalizar el final de un bloque, se usa la Sentencia [FIN](#).

Todas las sentencias de Bloque siguen una extensión de este comportamiento. Incluyendo el terminar con FIN.

BORRAR

Indicador de Sentencia » Persistencia

/ [Guía - Persistencia de Datos - Borrando Datos Guardados](#)

Sintaxis

```
GUARDAR <identificador> con <expresión>
GUARDAR <identificador>
```

Los < > indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Borra el valor bajo el identificador especificado de los datos guardados del Tubérculo, para que no se recupere como variable en la *siguiente ejecución* del Tubérculo que se está ejecutando. **NO BORRA VARIABLES DE LA EJECUCIÓN ACTUAL.**

Campo	Opcional	Descripción
Identificador		Nombre bajo el cual guardar el dato.

A partir de la ejecución posterior, cualquier variable que se haya guardado bajo el identificador indicado antes de esta sentencia ya no será cargado, pues no existirá.

Persistencia de Datos de Tubérculos

Los datos que guardas en tus Tubérculos se almacenan en la **base de datos persistente de Bot de Puré**, junto a tu código y las diferentes variantes de Entradas de Usuario. Por ende, los datos guardados no se borrarán a menos borres o sobre-escribas el Tubérculo.

CARGAR

Indicador de Sentencia » Asignación

/ [Guía - Gramática – Sentencias](#) / [Guía - Declaraciones](#)

Sintaxis

CARGAR <identificador> con <expresión>

CARGAR <identificador>

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

A un nivel básico, asigna el valor de la **expresión** proporcionada a la variable bajo el identificador especificado.

Campo	Opcional	Descripción
Identificador		Nombre de variable a declarar o actualizar.
Expresión	✓	Expresión a asignar.

La sentencia CREAR es altamente adaptativa y demuestra diferentes comportamientos dependiendo de un par de condiciones:

- **La variable existe y se suministra una expresión** — se asigna el resultado de evaluación de la **expresión** a la variable.
- **La variable existe y NO se suministra una expresión** — no ocurren cambios.
- **La variable NO existe y se suministra una expresión** — se *declara* la variable en el *ámbito actual* y se le da un valor correspondiente al resultado de evaluación de la **expresión** dada.
- **La variable NO existe y NO se suministra una expresión** — se *declara* la variable en el *ámbito actual* y se le asigna **Nada**.

Declaración Automática

Si no existe ninguna variable bajo dicho identificador, se la *declara* en el *ámbito actual*. Lo cuál demuestra un comportamiento idéntico a la Sentencia CREAR.

Asignación Condicional

Si no se indica una **expresión**, el valor asignado será el valor de la misma variable bajo el identificador indicado. O sea que si dicha variable **no** existe, se cumple el comportamiento de **declaración automática** y se le asigna el valor **Nada** a la variable declarada.

Ejemplos

El siguiente programa tiene una probabilidad de 10% de enviar un mensaje sobre el otro:

```
CARGAR prob con aleatorio(1)

CREAR Texto mensaje
SI prob precede 0.1
    CARGAR mensaje con "¡Bien! ¡Parece que tuviste suerte!"
SINO
    CARGAR mensaje con "Bueno... mejor suerte la próxima."
FIN

ENVIAR mensaje
```

El siguiente programa carga una canasta con frutas aleatorias según diga el Usuario:

```
LEER Número cantidad con 3
PARAR con "La canasta no tiene frutas :(" si cantidad no excede 0

CREAR Lista canasta
MIENTRAS cantidad excede 0 RESTAR cantidad
    //Se declara la Lista "frutas" en cada iteración
    CARGAR frutas con Lista
        "Manzana",
        "Naranja",
        "Banana",
        "Pera",
    CARGAR f con dado(frutas→largo)

    EXTENDER canasta con frutas→(f)
FIN

COMENTAR "La Lista 'frutas' ya no existe en este punto del código,
          o sea que se puede declarar una nueva variable 'frutas'"
CARGAR frutas con canasta→unir(", ")
ENVIAR frutas
```

COMENTAR

Indicador de Sentencia

/ [Guía - Gramática - Comentarios](#)

Sintaxis

COMENTAR <literal de Texto>

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Permite introducir un Texto a modo de comentario. No se evalúa.

Campo	Opcional	Descripción
Texto		Un literal de Texto con tu comentario.

Ejemplo

El siguiente código simplemente envía “Hace calor”.

COMENTAR "Puedes escribir lo que se te ocurra aquí dentro. Como recomendación, deberías comentar sobre lo que hace tu código cuando sientas que se lee muy confuso. Realmente no hay un límite preciso respecto a qué tan largo puede ser un comentario, así que..."

Tubérculo: Un tubérculo es un tallo subterráneo modificado y engrosado donde se acumulan los nutrientes de reserva para la planta (cumpliendo la función de órgano reservante).

Las especies que producen tubérculos también se sirven de ellos para propagarse en forma vegetativa, aunque sus semillas sean viables.

*Existen especies que producen tubérculos comestibles para el ser humano y por ello se cultivan desde hace milenios, en particular en América del Sur. La papa o patata (*Solanum tuberosum*), con origen en el altiplano sur del Perú, es el tubérculo más consumido en el mundo y se encuentra entre los diez principales cultivos de la humanidad. Otros tubérculos cultivados son la oca (*Oxalis* *tuberosa*), el ñame ("*

ENVIAR "Hace calor"

CREAR

Indicador de Sentencia

/ [Guía - Declaraciones](#)

Sintaxis

```
CREAR <identificador>
CREAR <identificador1>, <identificador2>
CREAR <identificador1>, <identificador2>, (...), <identificadorN>

CREAR <tipo> <identificador>
CREAR <tipo> <identificador1>, <identificador2>, (...), <identificadorN>
```

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Declara una variable bajo el identificador proporcionado y le asigna el *valor por defecto* del tipo especificado. Si se indican *múltiples* identificadores, se declaran *múltiples* variables con **ese mismo valor**. Si no se indica un tipo, se asigna [Nada](#) a cada valor en su lugar.

Campo	Opcional	Descripción
Tipo	✓	Tipo de la variable a declarar.
Identificador		Nombre bajo el cual se declara la variable.

Valores por defecto

- Número: `0`
- Texto: `""`
- Lógico: `Falso`
- Lista: `Lista` (Lista vacía)
- Registro: `Registro` (Registro vacío)
- Marco: (Marco vacío)

Si bien declarar múltiples variables al mismo tiempo les asigna el mismo valor, ten en cuenta que tienen **diferente identidad** y los cambios realizados en una de las variables no se verá reflejado en las demás incluso si el valor es una [estructura](#).

DEVOLVER

Indicador de Sentencia » Omisora

/ [Guía - Funciones - Retorno de Valores](#)

Sintaxis

```
DEVOLVER <expresión>
```

Los < > indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Termina la ejecución de la [Función](#) actual y **devuelve** la *evaluación* de la [expresión](#) dada.

Campo	Opcional	Descripción
Expresión		Valor a devolver.

Devolución de Valores en Funciones

Todas las Funciones devuelven un valor, y esta sentencia permite especificar qué devolver. Cualquier Función que no ejecute esta sentencia devolverá [Nada](#) en su defecto.

Una vez ejecutada, la Sentencia DEVOLVER **escapará** cualquier ámbito que no sea de Función hasta lograr que una Función devuelva el valor expresado.

En el caso de toparse con el ámbito del bloque Programa antes que un ámbito de Función, será el Tubérculo quien devolverá el valor indicado (esto no tiene ninguna aplicación práctica en esta versión de PuréScript).

En caso de solo querer terminar la ejecución de la Función sin la necesidad de devolver un valor, puedes simplemente devolver Nada, o usar la Sentencia [TERMINAR](#).

```
DEVOLVER Nada
```

Omisión de Sentencias

`DEVOLVER` omite el resto de sentencias hacia abajo en el cuerpo de la Función una vez ejecutado. Usarlo fuera de un ámbito de Función hará que se *aborte* el Tubérculo.

Si se usa dentro de una estructura iterativa, la misma dejará de iterar inmediatamente.

Todas las sentencias que se estén dentro del *mismo ámbito* y *luego* de esta sentencia serán ignoradas directamente por el analizador. Mientras tanto, aquellas que estén fuera del ámbito de ejecución pero dentro del ámbito seleccionado serán ignoradas activamente por el intérprete.

DIVIDIR

Indicador de Sentencia » Asignación

Sintaxis

```
DIVIDIR <identificador> con <expresión>
```

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Divide la variable numérica bajo el identificador indicado por la *representación numérica* de la *expresión* especificada.

Requiere que el identificador ya esté declarado.

Campo	Opcional	Descripción
Identificador		Nombre de una variable.
Expresión		Expresión por la cual dividir la variable.

Es una versión corta de:

```
CARGAR <identificador> con <identificador> / (<expresión>)
```

EJECUTAR

Indicador de Sentencia

/ [Guía - Funciones - Sentencia EJECUTAR](#)

Sintaxis

EJECUTAR <expresión>

USAR <expresión>

Los < > indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Evalúa una expresión. Útil para llamar [funciones](#) o métodos.

Campo	Opcional	Descripción
Expresión		Expresión a evaluar.

ENVIAR

Indicador de Sentencia

/ [Guía - Gramática - Sentencias](#)

Sintaxis

ENVIAR <expresión>

DECIR <expresión>

Los < > indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Envía un mensaje en Discord que corresponde a la expresión dada.

Campo	Opcional	Descripción
Expresión		Contenido a adjuntar al mensaje.

El mensaje no se envía inmediatamente, sino que se va acumulando con todas las sentencias ENVIAR hasta que el bloque Programa termina de ejecutarse, resultando en el envío de 1 solo mensaje. Cada uso de ENVIAR añade un renglón al mensaje enviado.

Tipos de Envío

Envío de Marco

Cuando la expresión se evalúa a un valor de tipo **Marco**, se enviará un Marco adjunto al mensaje cuyo diseño, campos y propiedades varias corresponden al valor.

CREAR Marco m

CARGAR título con "¡Buen estilo!"

CARGAR frase con "Este tipo de mensajes suele verse bien."

EJECUTAR m→agregarCampo(título, frase)

ENVIAR m

Envío de Otros Tipos

En cualquier otro caso, se intentará enviar la *representación textual* del valor. Si el tipo de valor no se pudiera convertir a Texto, se alzaría un error.

```
ENVIAR "Esto es un mensaje normalito, común, tradicional, etc."  
ENVIAR "Esto es... ¿otro mensaje? No realmente"  
ENVIAR "Hey " + usuario + ", bonita foto"  
  
ENVIAR Texto variableInexistente //Envía "Nada"  
ENVIAR Lista 1, 2, "hola", 3, Falso //Envía "(12hola3Falso)"  
ENVIAR Registro a: 1, b: "2", c: 3 //Envía "{Rg a: 1, b: 2, c: 3}"
```

Pila de Envíos

Enviar múltiples Textos hace que se pongan uno debajo del otro, en orden de envío. Todos en el mismo mensaje de Discord.

EXTENDER

Indicador de Sentencia » Asignación

Sintaxis

```
EXTENDER <identificador> con <expresión>
EXTENDER <identificador> en <índice> con <expresión>
```

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Añade un elemento a la Lista bajo el identificador indicado. El valor del elemento corresponde a la evaluación de la expresión otorgada.

Campo	Opcional	Descripción
Identificador		Nombre de una Lista.
Expresión		Expresión a asignar al elemento mencionado.
Índice	✓	índice dentro de la Lista seleccionada.

Indizado

Puedes agregar la palabra clave `en` para indicar un índice particular en lugar del último.

Sin “en”

Añade un elemento al final de la Lista, cuyo valor corresponde a la `<Expresión>` dada.

Con “en”

Añade un elemento en el índice especificado de la Lista.

- Si el índice es positivo, se empujan por +1 posición todos los elementos desde el índice (inclusive) especificado hasta el final de la Lista y se agrega el elemento en la posición que se indicó.
- Si el índice es negativo, la posición de inserción se cuenta desde el final de la Lista. Los elementos después de la posición calculada se empujan por +1 posición.

FIN

Indicador de Sentencia » Estructura de Control » Flujo

Sintaxis

```
BLOQUE
...sentencias
FIN
```

El texto oscurecido está a modo de ofrecer contexto y puede que no se escriba exactamente como está.

Termina el [bloque](#) actual, delimitando su alcance. Destruye el ámbito del bloque delimitado al ser ejecutada y, por consecuencia, las variables contenidas en este.

Nótese que no puedes terminar el bloque Programa de esta forma, ya que el Programador no tiene acceso explícito a este. Si quieres terminar el bloque Programa de forma temprana, puedes usar [TERMINAR](#), que omite el resto de sentencias en un bloque iterativo, bloque Programa o Función.

Estructuras Compatibles

Interacción con Otras Sentencias de Bloque

Las siguientes sentencias de bloque pueden ser delimitadas con la Sentencia FIN:

BLOQUE	MIENTRAS	PARA
SI	SINO	REPETIR

Interacción con Funciones

FIN también actúa como delimitador del cuerpo de una [expresión de Función](#):

```
Función(<argumento1>, <argumento2>, (...), <argumentoN>
...sentencias/cuerpo de Función
FIN
```

GUARDAR

Indicador de Sentencia » Persistencia

/ [Guía - Persistencia de Datos - Guardado de Datos](#)

Sintaxis

```
GUARDAR <identificador> con <expresión>
GUARDAR <identificador>
```

Los < > indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Guarda el valor de la **expresión** indicada bajo el identificador especificado para que sea accesible como variable en la *siguiente ejecución* del **Tubérculo** que se está ejecutando.

Campo	Opcional	Descripción
Identificador		Nombre bajo el cual guardar el dato.
Expresión	✓	Expresión a guardar.

A partir de la ejecución posterior, se declarará una variable con el mismo **identificador** y se le asignará el último valor guardado para el mismo.

Es importante aclarar que esta sentencia **no** asigna una variable al ser ejecutada. Lo que hace en realidad es asociar el identificador especificado al valor de la **expresión directamente dentro de la memoria del Tubérculo ejecutado**, con tal de que se asigne a una variable al ejecutar el programa otra vez.

Persistencia de Datos de Tubérculos

Los datos que guardas en tus **Tubérculos** se almacenan en la **base de datos persistente de Bot de Puré**, junto a tu código y las diferentes variantes de Entradas de Usuario. Por ende, los datos guardados no se borrarán a menos borres o sobre-escribas el **Tubérculo**.

HACER HASTA

Indicadores de Sentencia » Estructura de Control » Iterativa

Sintaxis

HACER

...sentencias a ejecutar 1 vez y Luego si la expresión es verdadera

HASTA <condición>

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido ofrece contexto.

HACER marca el inicio de un [bloque](#) que se ejecuta **hasta** que la [condición](#) dada se cumple (o sea que al ser convertida a su *representación lógica*, evalúa a Verdadero).

El bloque iniciado por HASTA no finaliza con [FIN](#). En su lugar, es delimitado por un HASTA seguido de la expresión de [condición](#) de corte.

Campo	Opcional	Descripción
Condición		Expresión a evaluar para determinar si dejar de iterar (Verdadero) o no (Falso) luego de la primer iteración.

HACER...HASTA tiene un comportamiento muy similar a [MIENTRAS](#). Las diferencias son que **primero se ejecuta el bloque** y luego se comprueba la [condición](#) a cumplir **para dejar de iterar** (no para seguir iterando) nuevamente. El bloque se ejecuta al menos una vez.

Bucles Infinitos

Si la [condición](#) nunca evalúa a un valor que permita dejar de iterar el bloque, se habla de un *bucle infinito*, lo cuál eventualmente causará un error por límite de sentencias procesadas. Siempre querrás asegurarte de que los contenidos del bloque iterativo hagan que la [condición](#) eventualmente permita un corte. Alternativamente, puedes usar [TERMINAR](#) dentro del bloque para cortar las iteraciones posteriores e interrumpir la iteración actual.

LEER

Indicador de Sentencia » Asignación

/ [Guía - Recibiendo Entradas de Usuario - Sentencia LEER](#)

Sintaxis

```
LEER <tipo> <identificador>
LEER <tipo> <identificador> con <respaldo>
LEER <tipo> opcional <identificador>
LEER <tipo> opcional <identificador> con <respaldo>
LEER <tipo> <identificador> <formato>
LEER <tipo> <identificador> con <respaldo> <formato>
LEER <tipo> opcional <identificador> <formato>
LEER <tipo> opcional <identificador> con <respaldo> <formato>
```

Los < > indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Permite interceptar y recibir una [Entrada de Usuario](#), convertirla al **tipo** indicado y asignarla al identificador especificado. Es una sentencia altamente ligada a la forma en la que se ejecuta el **Tubérculo** al que pertenece el programa.

Se puede especificar una expresión de **respaldo** para los casos en los que se esté realizando una **ejecución de prueba** (o **primera ejecución**) o los casos que *no* se reciba un dato de Entrada del Usuario y la misma se haya marcado como **opcional**.

Campo	Opcional	Descripción
Tipo		El tipo al cual se intentará convertir el dato a asignar.
Identificador		Nombre de variable a declarar o actualizar.
Respaldo	✓	Expresión de respaldo a evaluar y asignar si no se recibe una dato de parte del Usuario.
Formato	✓	Directriz de formato a aplicar a la Entrada.

La Sentencia LEER exhibe diferentes comportamientos dependiendo de si se usa la palabra clave **opcional**, si se pasa un valor de respaldo, y si se está realizando una **ejecución de prueba (primera ejecución)** o una **ejecución ordinaria**.

Diferencias Situacionales

Durante una *ejecución ordinaria*, **solo si** el Usuario ofrece un valor de Entrada, se asignará dicho valor a la variable bajo el identificador especificado, actuando prácticamente como un **CARGAR** cuya expresión es dada por quien ejecuta el Tubérculo.

Según Valor de Respaldo

La existencia o inexistencia del valor de **respaldo** **solo** importa cuando **no** se recibe un dato de parte del Usuario, y determina qué valor se le asignará en su lugar al identificador.

Si se provee una expresión de valor de **respaldo**, se le asigna ese valor en su lugar. De lo contrario, se asigna el *valor por defecto* del **tipo** de Entrada indicado.

Durante la [*ejecución de prueba*](#), siempre se trabaja con el valor de **respaldo o por defecto**.

Según Ejecución

Durante la *ejecución de prueba*, LEER **siempre** se comportará como si la Entrada fuera **opcional** y **no** hubiera recibido ningún valor del Usuario (y por ende se asignará el valor de **respaldo** o por defecto). En cambio, durante *ejecuciones posteriores* a que se haya guardado el Tubérculo, la sentencia *puede o no* recibir una Entrada de Usuario, en cuyo caso la respuesta dependerá de si se usó la palabra clave **opcional** o no.

Según Palabra Clave "opcional"

Si se usa o no la palabra clave **opcional** **solo** importa durante una *ejecución ordinaria o posterior al guardado*. Una Entrada opcional **evita** que se **alce un error** si el Usuario **no** ingresa un dato de Entrada, en cuyo caso se aplicaría lo explicado [arriba](#).

Declaración Automática

Si no existe ninguna variable bajo dicho identificador, se la *declara* en el *ámbito actual*. Esto demuestra un comportamiento idéntico a la Sentencia **CREAR**.

Directrices de Formato

Entradas de determinados tipos pueden ser configuradas con una **directriz de formato**.

Las *directrices de formato* dictan cómo se **transformará** el valor de Entrada recibido por la sentencia inmediatamente después de ser convertida al tipo indicado.

Las directrices de formato se escriben **siempre al final** de la sentencia.

Actualmente, PuréScript solo cuenta con directrices de formato para Números y Textos.

Tipo	Directriz	Sintaxis	Descripción
Número	Rango	entre A y B	Transforma el valor de Entrada al Número más cercano a este, dentro del rango descrito entre A y B <i>inclusive</i> . Si el valor ya estaba dentro del rango, no pasa nada.
Texto	Capitalizar	en minúsculas	Transforma el valor de Entrada de forma tal que todas las letras queden en minúsculas.
Texto	Capitalizar	en mayúsculas	Transforma el valor de Entrada de forma tal que todas las letras queden en mayúsculas.

MIENTRAS

Indicador de Sentencia » Estructura de Control » Iterativa

Sintaxis

```
MIENTRAS <condición>
...sentencias a ejecutar si la expresión es verdadera
FIN
```

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido ofrece contexto.

Marca el inicio de un [bloque](#) que se ejecuta **miéntras** la expresión de condición especificada pueda evaluarse a Verdadero.

Campo	Opcional	Descripción
Condición		Expresión a evaluar antes del bloque iterativo para decidir si seguir iterando (Verdadero) o no (Falso).

MIENTRAS es bastante similar a la Sentencia [SI](#), con la diferencia de que el bloque contenido puede ejecutarse *repetidas veces* en base a la [condición](#) en lugar de una sola.

En lugar de ejecutar el bloque que contiene y seguir con las sentencias debajo, esta sentencia repite su bloque una y otra vez mientras que la [condición](#) especificada siga teniendo una *representación lógica* equivalente a Verdadero.

Bucles Infinitos

Si la [condición](#) nunca evalúa a un valor que permita dejar de iterar el bloque, se habla de un *bucle infinito*, lo cuál eventualmente causará un error por límite de sentencias procesadas. Siempre querrás asegurarte de que los contenidos del bloque iterativo hagan que la [condición](#) eventualmente permita un corte. Alternativamente, puedes usar [TERMINAR](#) dentro del bloque para cortar las iteraciones posteriores e interrumpir la iteración actual.

MULTIPLICAR

Indicador de Sentencia » Asignación

Sintaxis

```
MULTIPLICAR <identificador> con <expresión>
```

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Multiplica la variable numérica bajo el identificador indicado por la *representación numérica* de la *expresión* especificada.

Requiere que el identificador ya esté declarado.

Campo	Opcional	Descripción
Identificador		Nombre de una variable.
Expresión		Expresión por la cual multiplicar la variable.

Es una versión corta de:

```
CARGAR <identificador> con <identificador> * (<expresión>)
```

PARA

Indicador de Sentencia » Estructura de Control » Iterativa

Sintaxis

```
PARA <contador> con <inicialización> MIENTRAS <condición> <actualización>
...sentencias a ejecutar si la expresión es verdadera
FIN
```

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido ofrece contexto.

Marca el inicio de un **bloque** que se ejecuta **mientras** la expresión de **condición** especificada pueda evaluarse a **Verdadero**. Además, provee la utilidades para trabajar con una *variable contador*. Existe un **PARA** corto con una sintaxis diferente, puedes verlo [aquí](#). La Sentencia PARA larga consiste de 3 pasos: *inicialización, comprobación y actualización*.

Paso de Inicialización

El *paso de inicialización* se ejecuta **una sola vez** al alcanzar el **PARA**, **antes** de ejecutar su bloque u otros pasos. Crea un ámbito en el cual *declara* la variable contador especificada y le asigna el valor de **inicialización** proveído. El bloque de la estructura termina siendo un ámbito **dentro** del creado en este paso, pero **no el mismo**. Dicho ámbito interno se crea y destruye por cada iteración, como con el resto de estructuras iterativas.

Paso de Comprobación

El paso de comprobación se comienza a realizar repetidamente **después** del *paso de inicialización*. Este paso determina si ejecutar el bloque iterativo o parar de iterar dependiendo de si la expresión de **condición** puede evaluar a **Verdadero**. La expresión dada tiene acceso a la variable contador declarada en el *paso de inicialización*.

Paso de Actualización

Después de cada ejecución del bloque iterativo, pero **antes** de volver a realizar el *paso de comprobación*, se ejecuta la sentencia de **actualización** detallada. Esta sentencia tiene acceso al ámbito creado en el *paso de inicialización*, y generalmente se la usa para

modificar el valor de la variable contador que fue declarada ahí mismo (de ahí por qué se llama “paso de actualización”).

Campo	Opcional	Descripción
Contador		Identificador de la variable contador a declarar en el ámbito del PARA antes de comenzar a iterar.
Inicialización		Expresión de valor inicial a asignar al identificador cuando se lo declara en el paso de inicialización.
Condición		Expresión a evaluar para determinar si realizar una iteración (Verdadero) o no (Falso).
Actualización		Sentencia a ejecutar luego de cada iteración.

PARA tiene un comportamiento similar a un MIENTRAS junto a una variable contador. Una diferencia a esto es que la variable contador es declarada “dentro del ámbito de la sentencia PARA”, por lo que será destruida al dejar de iterar:

```
CARGAR i con 0
MIENTRAS i precede 42
    SUMAR i //Esto se ejecuta 42 veces con diferentes valores de "i"
FIN
ENVIAR i //Envía "42"
```

```
PARA i con 0 MIENTRAS i precede 42 SUMAR i
    //Esto se ejecuta 42 veces con diferentes valores de "i"
FIN

ENVIAR Texto i //Envía "Nada"
```

Bucles Infinitos

Si la condición nunca evalúa a un valor que permita dejar de iterar el bloque, se habla de un *bucle infinito*, lo cuál eventualmente causará un error por límite de sentencias procesadas. Siempre querrás asegurarte de que los contenidos del bloque iterativo hagan que la condición eventualmente permita un corte. Alternativamente, puedes usar TERMINAR dentro del bloque para cortar las iteraciones posteriores e interrumpir la iteración actual.

PARA (Corto)

Indicador de Sentencia » Estructura de Control » Iterativa

Sintaxis

```
PARA <contador> desde <inicio> hasta <fin>
...sentencias a ejecutar con el rango de valores
FIN
```

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido ofrece contexto.

Permite declarar una variable contador cuyo valor por iteración corresponderá al rango de valores numéricos enteros entre la representación numérica de las expresiones de **inicio** y **fin** inclusive.

El valor que se le asigna inicialmente a la variable contador (y por ende su valor en la primer iteración) está dado por la expresión de **inicio**. Mientras tanto, la *dirección de actualización* del contador se determina según la relación entre los valores de **inicio** y **fin** evaluados, recorriendo de menor a mayor si **inicio** es menor que **fin** y de mayor a menor si **inicio** es mayor que **fin**. Esto significa que el contador siempre se dirigirá progresivamente desde el valor de **inicio** al valor de **fin**, en intervalos de ± 1 .

Campo	Opcional	Descripción
Contador		Identificador de la variable contador a declarar en el ámbito del PARA antes de comenzar a iterar.
Inicio		Expresión de valor inicial que se asignará al contador cuando se lo declare.
Fin		Expresión de valor al que se dirigirá progresivamente el contador designado.

Al igual que con el PARA largo, el PARA corto crea un ámbito propio en el cual declara la variable contador, y dicho ámbito **contiene** al ámbito del bloque iterativo designado, pero **no son el mismo ámbito**. Uno está dentro de otro y el interno se crea y destruye en cada iteración.

PARA CADA

Indicador de Sentencia » Estructura de Control » Iterativa

Sintaxis

PARA CADA <identificador> en <contenedor>

...esto se ejecuta una vez por cada elemento dentro del <contenedor>.

<identificador> se referirá a dicho elemento en cada iteración

FIN

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido ofrece contexto.

Permite iterar fácilmente sobre un **contenedor** con un identificador que corresponde al miembro que está siendo iterado actualmente.

Nótese que si iteras sobre una Lista, no tendrás acceso al índice de la iteración actual, o sea que **no** es recomendable usar **PARA CADA** si, además del elemento, necesitas saber el índice. La ventaja cuando se lo puede usar es que suele simplificar bastante el iterar sobre un **contenedor**.

Campo	Opcional	Descripción
Identificador		Nombre de la variable correspondiente al miembro de la iteración actual.
Contenedor		Expresión que evalúa a un contenedor iterable, como una Lista o Registro.

Identificador de Miembro Iterado

En **cada iteración del bloque designado**, se declara una variable en *el ámbito del mismo* bajo el identificador especificado. Su valor corresponde al miembro que se está iterando actualmente dentro del **contenedor**.

El identificador que se crea en cada iteración representa una **copia** del valor más que una **referencia** al mismo, así que **asignarle** un valor **no** reflejará cambios en el **contenedor** original (para eso, tendrías que usar un **PARA** común). En cambio, **modificar** el valor **sí** se reflejará en la Lista original (por ejemplo, una asignación de flecha).

Iteración sobre una Lista

Si el **contenedor** es una **Lista**, el identificador se referirá al valor de uno de sus elementos por cada iteración. Las Listas siempre se iteran en orden de inicio a fin.

```
CARGAR unaLista con Lista 0, 1, 2  
//Envía el valor de todos los elementos con un formato  
PARA CADA elemento en unaLista  
    ENVIAR "Este elemento vale: " + elemento  
FIN
```

Aquí se demuestra la diferencia entre asignar y modificar `elemento`:

```
// --- ASIGNACIÓN ---  
CARGAR unaLista con Lista 0, 1, 2  
  
PARA CADA elemento en unaLista  
    CARGAR elemento con elemento * 2  
FIN  
  
ENVIAR unaLista→unir(", ") //Envía "0, 1, 2"  
  
// --- MODIFICACIÓN ---  
//Nota: esto es una Lista de Listas  
CARGAR unaLista con Lista (Lista 0), (Lista 1), (Lista 2)  
  
PARA CADA elemento en unaLista  
    CARGAR elemento→0 con elemento→0 * 2  
FIN  
  
ENVIAR unaLista→unir(", ") //Envía "(0), (2), (4)"
```

Iteración sobre un Registro

Si el **contenedor** es un **Registro**, el identificador se referirá a **una Lista de 2 elementos** que *represente* una de las entradas del mismo por cada iteración. La Lista contiene esto:

0. La **clave** de la Entrada, como un Texto.
1. El **valor** de la Entrada.

El orden en el que se itera el Registro **no** está garantizado.

Por dar un ejemplo sencillo, veamos el siguiente programa:

```
CARGAR unRegistro con Registro a: 42, b: "Papas", c: Falso  
  
//Envía lo siguiente:  
///"El valor de a es: 42!!!"  
///"El valor de b es: Papas!!!"  
///"El valor de c es: Falso!!!"  
PARA CADA entrada en unRegistro  
    ENVIAR "El valor de " + entrada→0 + " es: " + entrada→1 + "!!!"  
FIN
```

PARAR

Indicador de Sentencia » Omisora

/ [Guía - Recibiendo Entradas de Usuario - Limitando Entradas de Usuario](#)

Sintaxis

PARAR con <mensaje> si <condición>

PARAR con <mensaje>

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Si la condición expresada puede evaluarse a Verdadero, **aborta** la ejecución del programa, enviando **sólo** el mensaje especificado en lugar de cualquier otro mensaje que haya sido preparado con [ENVIAR](#).

Campo	Opcional	Descripción
Mensaje		El mensaje de aborto de ejecución.
Condición	✓	Expresión a evaluar para determinar si abortar la ejecución del programa (Verdadero) o no (Falso).

Si no es un Texto de antemano, la expresión de mensaje será convertida a una representación textual antes de enviarse (si es que se envía).

Esta sentencia es muy útil para limitar el rango de valores que un [Usuario](#) puede ingresar como Entrada de Usuario. Por este motivo, se suele usar luego de la Sentencia [LEER](#):

LEER Número nro

//Evita que el Usuario ingrese valores menores que 0
PARAR con "Por favor, ingresa 0 ó más" si nro precede 0

//Evita que el Usuario ingrese valores mayores a 100
PARAR con "Por favor, ingresa 100 ó menos" si nro excede 100

CARGAR formateado con (10 ^ nro)→formatear(Verdadero, 1)
ENVIAR "A un 1 seguido de " + nro + " ceros se le dice: " + formateado

PARAR sin condición

PARAR puede ser utilizado para abortar el Tubérculo cuando se lo ejecute sin necesidad de comprobar una condición. Esto difiere de la Sentencia TERMINAR en que el efecto de terminación escapará cualquier ámbito de Función y continuará hasta que el programa se detenga por completo. Para usar PARAR sin condición, omite el `si` y la condición.

Omisión de Sentencias

PARAR omite el resto de sentencias hacia abajo en *el ámbito seleccionado* del listado. Una vez ejecutada la sentencia, no se inician nuevos ámbitos hasta escapar del seleccionado.

Dada la naturaleza condicional de esta sentencia, el analizador sintáctico no remueve automáticamente ninguna de las sentencias posteriores a esta. En cambio, es el intérprete el que ignora activamente cualquier sentencia posterior en caso de aborto de ejecución.

La normativa de omisión **escapa** cualquier ámbito de Función o iteración, y no deja de omitir sentencias hasta que el Tubérculo deje de ejecutarse.

REPETIR

Indicador de Sentencia » Estructura de Control » Iterativa

Sintaxis

REPETIR <cantidad> veces

...sentencias a ejecutar si la expresión es verdadera

FIN

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido ofrece contexto.

Marca el inicio de un **bloque** que se ejecuta, como máximo, la **cantidad** de veces indicada. El valor de la expresión de **cantidad** es evaluado *una sola vez*, **antes** de comenzar a iterar.

Campo	Opcional	Descripción
Cantidad		Cantidad de veces a iterar el bloque como máximo.

Interacción con Sentencia TERMINAR

Al igual que el resto de estructuras iterativas: si se usa **TERMINAR** dentro de su bloque, el mismo dejará de iterarse sin importar qué. Para **REPETIR** esto significa que el bloque podría llegar a ejecutarse menos veces que lo indicado por la expresión de **cantidad**.

Si no se usa **TERMINAR**, esto obviamente no aplica, así que **REPETIR** se ejecutaría exactamente la **cantidad** de veces indicada.

Bucles Infinitos

Si la **condición** nunca evalúa a un valor que permita dejar de iterar el bloque, se habla de un *bucle infinito*, lo cuál eventualmente causará un error por límite de sentencias procesadas. Siempre querrás asegurarte de que los contenidos del bloque iterativo hagan que la **condición** eventualmente permita un corte. Alternativamente, puedes usar **TERMINAR** dentro del bloque para cortar las iteraciones posteriores e interrumpir la iteración actual.

RESTAR

Indicador de Sentencia » Asignación

Sintaxis

```
RESTAR <identificador>
RESTAR <identificador> con <expresión>
```

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Resta un Número a la variable **numérica** bajo el identificador especificado. Si se da una expresión, se resta ese valor. Si no, se resta **1**.

Requiere que el identificador ya esté declarado.

Campo	Opcional	Descripción
Identificador		Nombre de una variable.
Expresión	✓	Expresión a restarle a la variable.

Sin Expresión

Es una versión corta de:

```
CARGAR <identificador> con <identificador> - 1
```

Con Expresión

Es una versión corta de:

```
CARGAR <identificador> con <identificador> - (<expresión>)
```

SI

Indicador de Sentencia » Estructura de Control » Condicional

/ [Guía - Bloques y Estructuras de Control - Estructuras Condicionales](#)

Sintaxis

```
SI <condición>
    ...sentencias a ejecutar si la condición se cumple
FIN
```

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido ofrece contexto.

Marca el inicio de un **bloque** que se ejecuta si la **condición** se puede evaluar a **Verdadero**.

Campo	Opcional	Descripción
Condición		Expresión a evaluar para determinar la ejecución u omisión del bloque contenido.

Si la evaluación de la expresión de **condición** dada no es de tipo Lógico, el valor se **convertirá** a su *representación lógica*.

```
CARGAR a con 42

SI a excede 20
    //Esto se ejecuta porque 42 es mayor que 20
FIN

SI a precede 30
    //Esto NO se ejecuta porque 42 NO es menor que 30
FIN

SI a
    //Esto se ejecuta porque 42 convertido a Lógico es Verdadero
FIN

SI b
    //Esto NO se ejecuta porque Nada convertido a Lógico es Falso
FIN
```

Puedes ver las conversiones a Lógico [aquí](#).

SINO

Indicador de Sentencia » Estructura de Control » Condicional

/ [Guía - Bloques y Estructuras de Control - Estructuras Condicionales](#)

Sintaxis

```
SI <condición>
    ...sentencias a ejecutar si la expresión es verdadera
SINO
    ...sentencias a ejecutar si la expresión es falsa
FIN
```

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido ofrece contexto.

Reemplaza la [Sentencia FIN](#) que delimita una [Sentencia SI](#). Delimita el bloque de la misma en su lugar y marca el inicio de otro [bloque](#) a continuación. Esta sentencia **solo** puede ser usada como reemplazo del [FIN](#) de un [SI](#), o sea que *no puede existir por si sola*.

El bloque iniciado **solo** se ejecuta si no se cumple la condición del [SI](#) delimitado. En otras palabras: [SINO](#) inicia un bloque que se ejecuta **solo** cuando no se ejecuta el bloque por encima del mismo. Esto significa que si se cumple la condición, se ejecuta el bloque del [SI](#), y si no se cumple la condición, se ejecuta el bloque del [SINO](#) que está a continuación.

Estructura "SI...SINO...SI"

Puedes “concatenar” múltiples comprobaciones lógicas de la siguiente manera:

```
SI a
    //Esto se ejecuta si "a" es Verdadero...
SINO SI b
    //Esto se ejecuta si "a" es Falso y "b" es Verdadero...
SINO SI c
    //Esto se ejecuta si "a" y "b" son Falso y "c" es Verdadero...
SINO
    //Esto se ejecuta si "a", "b" y "c" son Falso...
FIN
```

Como verás, cada [SINO](#) subsecuente solo se ejecuta si ningún bloque anterior lo hace.

SUMAR

Indicador de Sentencia » Asignación

/ [Guía - Gramática - Sentencias](#)

Sintaxis

```
SUMAR <identificador>
SUMAR <identificador> con <expresión>
```

Los <> indican plantillas reemplazables, mientras que el resto del texto debe escribirse igual. El texto oscurecido no se escribe.

Suma un valor a la variable bajo el identificador especificado. La variable **debe** ser un **Número** o un **Texto**, y se realizan diferentes operaciones dependiendo del *tipo*:

- Si la variable es un **Texto**, se realiza una concatenación.
- Si la variable es un **Número** y la **expresión** es un **Texto**, se realiza concatenación.
- Si la variable es un **Número** y la **expresión** **no** es un **Texto**, se realiza una suma.

Requiere que el identificador ya esté declarado.

Campo	Opcional	Descripción
Identificador		Nombre de una variable.
Expresión	✓ ¹	Expresión a sumarle a la variable.

Sin Expresión

Esta forma de SUMAR solo funciona con Números. Intentar sumar un Texto sin proporcionar una expresión **alzará un error**. Dicho esto, si la variable no es de tipo Texto, **SUMAR sin expresión** es solo una versión corta de:

```
CARGAR <identificador> con <identificador> + 1
```

¹ Solo es opcional si la variable mencionada es un Número.

Con Expresión

Es una versión corta de:

```
CARGAR <identificador> con <identificador> + (<expresión>)
```

Nótese que incluso aplican las mismas reglas descritas para distinguir entre el operador `+` de concatenación textual y el operador `+` de suma aritmética.

Esto significa que si el identificador o la expresión son Textos, entonces se realizará una concatenación. Si ninguno de los dos es un Texto, entonces se realiza una suma aritmética.

TERMINAR

Indicador de Sentencia » Omisora

/ [Guía - Bloques y Estructuras de Control - Otras Sentencias de Control](#)

Sintaxis

TERMINAR

En la práctica, ejerce uno de tres comportamientos distintos dependiendo del *ámbito* en el que se ejecuta (solo se considera el *ámbito* más cercano/anidado/específico de entre los listados a continuación):

- **Ámbito de Función** — termina la ejecución de la [Función](#) actual.
- **Ámbito de bloque iterativo** — termina inmediatamente la ejecución del bloque iterativo y detiene la iteración.
- **Ámbito de bloque Programa** — termina la ejecución del *Tubérculo* actual.

En un sentido más técnico, un `TERMINAR` ejecutado **omitirá** sentencias y **escapará** ámbitos hasta toparse *directamente* con uno de los ámbitos descritos arriba y escaparlo también.

Si se topa primero con un *ámbito* de estructura iterativa, entonces detendrá su iteración inmediatamente, independientemente de si su condición de iteración aun se cumple.

Omisión de Sentencias

`TERMINAR` omite el resto de sentencias hacia abajo en el *ámbito seleccionado* del listado.

Una vez ejecutada la sentencia, no se inician nuevos *ámbitos* hasta escapar del seleccionado.

Todas las sentencias que se estén dentro del *mismo ámbito* y *luego* de esta sentencia serán ignoradas directamente por el analizador. Mientras tanto, aquellas que estén **fueras** del *ámbito de ejecución* pero **dentro** del *ámbito seleccionado* serán ignoradas activamente por el intérprete.

Número

Tipo de valor primitivo

Representa un valor numérico en el dominio de los números reales. Comúnmente devuelto por operaciones aritméticas.

Internamente, los valores de tipo numérico se guardan como un número de punto flotante de doble presión que ocupa 64 bits. Esto significa que pueden ocupar cualquier valor dentro del rango $-2^{53} + 1 \leq x \leq 2^{53} - 1$.

Expresión Literal

(Guía). Puedes expresar un *valor entero* si no utilizas un separador decimal. En cambio, para expresar un *número decimal* debes usar un punto (.) como separador. Si expresas un valor decimal, puedes omitir la parte entera, en cuyo caso esta será 0.

Los ceros por delante de la parte entera no afectan la cantidad. Cualquier carácter **después** del primero y **antes** del separador decimal puede ser un guion bajo (_). Este **no** afecta el valor evaluado, y puede ser usado como separador de miles o como se deseé.

```
3          //= 3
010        //= 10
123456789.33 //= 123,456,789.33
123_456_789.330 //= 123,456,789.33
.5          //= 0.5
```

Valor por Defecto

0

Conversiones de Número

- A Texto: Texto 3 = "3". Texto 3.14 = "3.14"
- A Lógico: Falso si el Número es 0; Verdadero en cualquier otro caso
- Desde Texto: Número "3" = 3; Número "a" = 0
- Desde Lógico: Número Verdadero = 1; Número Falso = 0

Números Negativos

No existe una expresión literal para representar un número negativo, sin embargo, puedes usar el [operador unario aritmético `-`](#) para **negar** un valor numérico:

```
-3          //= -3
-010        //= -10
-123456789.33 //= -123,456,789.33
-123_456_789.330 //= -123,456,789.33
-.5         //= -0.5
```

Miembros de Número

Puedes ver todos los métodos de Número [aquí](#).

Texto

Tipo de valor primitivo

/ [Guía - Gramática – Análisis Léxico e Interpretación](#)

Representa un valor textual. Es una secuencia ordenada de 0 ó más caracteres.

Expresión Literal

Siempre comienza y termina con comillas dobles ("), las cuales no forman parte del valor evaluado en sí. Las *mayúsculas*, *minúsculas* y *tildes* escritas dentro de las comillas **se ven reflejadas exactamente igual** en el valor evaluado. Puede tener [secuencias de escape](#).

```
"Puré de Papa"  
"¡Hola! ¿Andamos bien?"  
"42"  
"verdadero"  
"Nada"
```

Valor por Defecto

""

Conversiones de Texto

- A Número: Número "3" = 3; Número "a" = 0
- A Lógico: Falso si el texto está vacío; Verdadero en cualquier otro caso
- A Lista: "abc123"→aLista() = Lista "a", "b", "c", "1", "2", "3"
- Desde Número: Texto 3 = "3". Texto 3.14 = "3.14"
- Desde Lógico: Texto Falso = "Falso"; Texto Verdadero = "Verdadero"
- Desde Lista: Texto Lista = "(); Texto (Lista 1, "a", Falso) = "(1aFalso)"
- Desde Registro: Texto Registro = "{Rg}"; Texto (Registro a: "b") = "{Rg a: b }"
- Desde Marco: Texto <marco> = "[Marco]"
- Desde Función: Texto <fn.> = "[Función]"; Texto <f. nativa> = "[Función nativa]"
- Desde Nada: Texto Nada = "Nada"

Miembros de Texto

Todos los Textos tienen un miembro `Texto→largo` que indica automáticamente su cantidad de caracteres actual.

Puedes ver todos los métodos de Texto desde [aquí](#).

Lógico

Tipo de valor primitivo

Representa uno de dos valores lógicos: Verdadero o Falso. Suele ser el resultado de una [expresión lógica](#) más que un valor asignado de forma literal. Su principal función es la de *determinar condiciones en estructuras de control*, como lo serían aquellas iniciadas por las sentencias [SI](#), [MIENTRAS](#), [PARA](#), etc.

Expresiones Literales

(Guía). Puedes expresar un Lógico de forma literal escribiendo el nombre de cualquiera de sus valores: “verdadero” o “falso”. **No distinguen mayúsculas, minúsculas ni tildes** (pero en el código de este documento siempre se pone en mayúsculas la primer letra).

Verdadero
Falso

Valor por Defecto

Falso

Conversiones de Lógico

- A [Número](#): Número Verdadero = 1; Número Falso = 0
- A [Texto](#): Texto Falso = "Falso"; Texto Verdadero = "Verdadero"
- Desde [Número](#): Falso si el Número es 0; Verdadero en cualquier otro caso
- Desde [Texto](#): Falso si el texto está vacío; Verdadero en cualquier otro caso
- Desde [Lista](#): Falso si la Lista está vacía; Verdadero en cualquier otro caso
- Desde [Registro](#): Falso si el Registro está vacío; Verdadero en cualquier otro caso
- Desde [Marco](#): Lógico <marco> = Verdadero
- Desde [Función](#): Lógico <función> = Verdadero
- Desde [Nada](#): Lógico Nada = Falso

Lista

Tipo de valor estructural

Representa un conjunto *ordenado* de valores, denominados “elementos”. Dichos elementos pueden ser de **cualquier** tipo: [Números](#), [Textos](#), incluso **otras Listas**, entre otros.

Los elementos son indexados **desde 0** en adelante. O sea que el primer elemento se encontrará en el índice 0 y el último índice equivale a *cantidadDeElementos* – 1.

Las Listas de PuréScript son **heterogéneas**, lo cual significa que sus elementos pueden ser de diferentes tipos, pero nada te impide popularlas de forma *homogénea*.

Expresión Literal

([Guía](#)). Para expresar una Lista, se usa el [indicador de tipo Lista](#) seguido de expresiones separadas por comas. El valor evaluado de cada una de dichas expresiones se enlaza a la Lista como un elemento. **El orden de las expresiones importa**, la primera es el índice 0.

A continuación puedes ver:

1. Una Lista vacía
2. Una Lista de Números del 1 al 9 (indizados de 0 a 8)
3. Una Lista con elementos de varios tipos, conteniendo incluso otra Lista en el índice 4 y un [Registro](#) en el índice 5

```
//1
Lista

//2
Lista 1, 2, 3, 4, 5, 6, 7, 8, 9

//3
Lista 42, "Hola", Verdadero, (Lista 1 + 2, Falso), (Registro a: 3)
```

Coma final ignorable

Si pones una coma después del último elemento y antes de una nueva sentencia, esta coma será ignorada. Esto suele ser más cómodo y consistente:

```
CARGAR li con Lista  
1,  
2,  
3,  
  
ENVIAR li→unir(", ") //Envía "1, 2, 3"
```

Comas extra

Si pones comas extra entre elementos, o más de una coma al final, se añadirán espacios que valen [Nada](#) entre las *comas adyacentes*:

```
CARGAR li con Lista 1, 2, 3, , 5, , ,  
//Envía "1.2.3.Nada.5.Nada.Nada"  
ENVIAR li→unir(".")
```

Acceso a elementos con operador de flecha

Para acceder a los elementos de una Lista, se emplea el [operador de flecha](#) (\rightarrow). Este es un operador binario cuyo operando izquierdo es la Lista de la cual se accede y cuyo operando derecho es el índice a acceder.

Intentar acceder un índice que no existe en la Lista evaluará a [Nada](#).

Valor por Defecto

Una lista vacía. O sea, sin elementos (y por ende, sin índices accesibles).

Conversiones de Lista

- A Texto: `Texto Lista = "()"; Texto (Lista 1, "a", Falso) = "(1aFalso)"`
- A Lógico: `Falso` si la Lista está vacía; `Verdadero` en cualquier otro caso
- A Registro: `(Lista "a", "b", "c")→aRegistro() = Registro 0: "a", 1: "b", 2: "c"`
- Desde Texto: `"abc123"→aLista() = Lista "a", "b", "c", "1", "2", "3"`

Miembros de Lista

Todas las Listas tienen un miembro `Lista→largo` que indica automáticamente su cantidad de elementos actual.

Puedes ver todos los métodos de Lista desde [aquí](#).

Registro

Tipo de valor estructural

Representa un conjunto de pares relacionados de claves y valores, denominados “entradas”, similar a cómo un diccionario contiene pares relacionados de palabras y definiciones. A diferencia de las [Listas](#), **el orden de las entradas no está garantizado**.

La clave de una entrada debe ser un [identificador](#) o un [literal de número](#), mientras que su valor asociado puede ser un [Número](#), [Texto](#) o incluso [otro Registro](#), entre otros.

Expresión Literal

(Guía). Para expresar un Registro, se usa el indicador de tipo [Registro](#), seguido de una sucesión de identificadores y [expresiones](#) (en ese orden) unidos por el [operador de dos puntos](#) (:) y separados por comas (,).

Cada par de identificador-expresión representa una [entrada](#), o sea que, en otras palabras, se escribe una *sucesión de entradas* separadas por comas.

A continuación puedes ver 3 ejemplos diferentes de expresiones literales de Registros:

```
Registro a: b, c: d, e: f  
Registro  
    g: "Hola",  
    h: Verdadero,  
    i: (Lista "caf ", (Registro t : 60, mocha: "vian s"), Falso),  
    j: 333  
Registro 100: 10 * 10, negativo: -20, 42: Nada
```

At m icamente, estas son las 2 formas de expresar literalmente una entrada:

```
<identificador>: <expresi n>  
<literal num rico>: <expresi n>
```

N tese que solo puedes expresar entradas dentro de expresiones de Registro.

Coma final ignorable

Si pones una coma después de la última entrada y antes de una nueva sentencia, esta coma será ignorada. Esto suele ser más cómodo y consistente:

```
CARGAR rg con Registro
a: 1,
b: 2,
c: 3,
d: 4,
ENVIAR rg->claves() ->unir(", ") //Envía "a, b, c, d"
ENVIAR rg->valores()->unir(", ") //Envía "1, 2, 3, 4"
```

Acceso a entradas con operador de flecha

Para acceder a los elementos de un Registro, se emplea el [operador de flecha](#) (\rightarrow). Este es un operador binario cuyo operando izquierdo es un contenedor del cual se accede (en este caso un Registro) y cuyo operando derecho es la clave a acceder.

Intentar acceder una clave que no existe en el Registro evaluará a [Nada](#).

Valor por Defecto

Un registro vacío. O sea, sin miembros (y por lo tanto, sin claves accesibles).

Conversiones de Registro

- A [Texto](#): `Texto Registro = "{Rg}"; Texto (Registro a: "b") = "{Rg a: b }"`
- A [Lógico](#): [Falso](#) si el Registro está vacío; [Verdadero](#) en cualquier otro caso
- Desde [Lista](#): `(Lista "a", "b", "c")->aRegistro() = Registro 0:"a", 1:"b", 2:"c"`
- Desde [Marco](#): `<marco>->aRegistro() = Registro Estándar de Marco`

Miembros de Registro

Todos los Registros tienen un miembro `Registro->tamaño` que indica automáticamente su cantidad de entradas actual. `Registro->largo` también es válido y hace lo mismo. No se consideran los *miembros de tipo* (como `tamaño`, `largo` y métodos nativos) para la cantidad.

Puedes ver todos los métodos de Registro desde [aquí](#).

Marco

Tipo de valor estructural especial

/ [Guía - Trabajando con Marcos](#)

Representa un marco embebido de mensaje (Embed) de **Discord**. Se podría decir que es una variación de los [Registros](#) que solo acepta unas propiedades particulares, siendo estas las necesarias para enviar y mostrar un marco.

Este tipo de valor solo debería ser usado para enviarse en un mensaje con la Sentencia [ENVIAR](#). Es una estructura bastante limitada y de poca utilidad en otras situaciones.

Las propiedades de un Marco no pueden ser accedidas por medio del [operador de flecha](#) (\rightarrow), ni tampoco pueden ser modificadas con el mismo.



Puedes leer las limitaciones de los valores de Marco en la [Guía](#).

Para modificar un marco, debes usar [métodos nativos de Marco](#).

Expresión Literal

Ninguna. Solo puedes declarar un Marco vacío con la Sentencia [CREAR](#).

Valor Inicial

Un Marco sin datos asociados. En este estado, el Marco **no** es enviable.

Conversiones de Marco

- A [Texto](#): `Texto <marco> = "[Marco]"`
- A [Lógico](#): `Lógico <marco> = Verdadero`
- A [Registro](#): `<marco> → aRegistro() = Registro Estándar de Marco`

Propiedades de Marco

Un Marco se asocia internamente a un conjunto de *propiedades privadas* para modificar la forma en la que se muestra en Discord.

Puedes modificar las propiedades de un Marco por medio de [métodos nativos de Marco](#).

Siéntete libre de usar esta imagen de referencia para tener más claro cómo diseñar un Marco.



Puedes ver todas las propiedades asociadas a sus respectivos métodos en esta tabla:

Propiedad	Método Accesor	Descripción
campos	→agregarCampo()	Lista de pares nombre-valor. Debajo del título/descripción y por encima del pie. Pueden haber hasta 3 por fila y 25 por marco.
autor	→asignarAutor()	Texto por encima del resto de componentes del Marco. Puede incluir un avatar.
autor→nombre	→asignarAutor()	El nombre del autor.
autor→avatar	→asignarAutor()	El avatar del autor. A la izquierda del nombre.
color	→asignarColor()	El <u>color</u> del borde izquierdo del Marco.
descripción	→asignarDescripción()	Descripción justo debajo del título del Marco.
enlace	→asignarEnlace()	Un enlace. Accesible por medio del título.
imagen	→asignarImagen()	Un enlace a una imagen. Justo por encima del pie y debajo de los campos del Marco.
miniatura	→asignarMiniatura()	Un enlace a una imagen. Arriba y a la derecha del resto de componentes del marco.
pie	→asignarPie()	Un Texto debajo del resto de componentes del Marco. Puede contener una imagen.
pie→texto	→asignarPie()	El Texto del pie de Marco.
pie→imagen	→asignarPie()	Una imagen a la izquierda del Texto del pie.
título	→asignarTítulo()	Un Texto justo por debajo del autor.

Función

Tipo de valor de procedimiento invocable

Representa una sucesión de sentencias que puede ser llamada o invocada y recibir valores externos para modificar su comportamiento. Los valores que se reciben externamente se denominan **argumentos**, y pasan a ser variables. Existen varias Funciones nativas.

Expresión literal

Para expresar una Función, se usa el indicador de tipo **Función**, seguido de los identificadores de sus argumentos entre paréntesis y separados por coma. Luego, se especifica el **cuerpo** de la Función.

El **cuerpo** de la Función contiene todas las **sentencias** a ejecutar cuando se la llame o invoque. Va desde el *cierre* de paréntesis de los argumentos hasta un **FIN** que no le corresponda a algún bloque dentro del cuerpo.

```
//Plantilla
Función(argumentos...)
    ...sentencias
FIN

//Ejemplos
Función() ENVIAR "¡Hola!" FIN

Función(a, b)
    DEVOLVER "El resultado es: " + (a + b)
FIN

Función(li) SUMAR li→el FIN

Función(a)
    CARGAR b con 10 + raíz(a, 2)
    CARGAR c con a * b

    SI c excede 1000
        DEVOLVER 1000
    FIN

    DEVOLVER c
FIN
```

Expresión Lambda

Puedes expresar una Función que simplemente recibe argumentos y devuelve una expresión de forma compacta y simplificada utilizando una **expresión Lambda**.

A diferencia de la expresión *completa* de Función vista arriba, la expresión Lambda no usa el **indicador de tipo Función** ni necesita ser delimitado con un **FIN**. En cambio, comienza con un identificador o una **secuencia** de identificadores.

```
//Plantilla
(argumentos...) ⇒ <expresión>

//Ejemplos
() ⇒ 42

(a, b) ⇒ "El resultado es: " + (a + b)

(x) ⇒ (x excede 1000 y 1000) o (x precede 0 y 0) o (x / 2 + 1)

x ⇒ x * 2 - 1
```

Tanto la expresión completa como la expresión Lambda evalúan a un valor de Función. Una diferencia clave entre funciones Lambda y funciones completas es que las funciones Lambda **no** disponen de un [ámbito de Función](#), y en su lugar tienen un [ámbito de bloque](#).

Valor por defecto

N/A

Conversiones de Función

- A [Texto](#): `Texto () ⇒ Nada` = "[Función]"; `Texto <f. nativa>` = "[Función nativa]"
- A [Lógico](#): `Lógico () ⇒ Nada` = [Verdadero](#)

Métodos

Un método es una **Función** que puede ser *accedida por medio de un valor* (generalmente por medio de una [estructura](#), pero también puede ser un [primitivo](#) para métodos de tipo). Estas funciones tienen nombres tanto relacionados a la acción que hacen como contextualizados al tipo u identidad del valor que las contiene.

Puedes definir tus propios métodos cuando trabajas con [Registros](#) (en esta versión de PuréScript, no hay manera de referirse a quien contiene una Función). Simplemente asigna un valor de Función a una clave del mismo. Nótese que también puedes asignar valores de Función a **índices** de [Lista](#) pero estos *no se suelen considerar métodos* por ser indizados.

Un ejemplo recurrente de métodos es el método de tipo de [Lista](#): `Lista → unir(separador)`, que une todas las *representaciones textuales* de sus elementos en un solo Texto con el `separador` especificado entre cada una:

```
(Lista 33, "AbC", Falso) → unir(" // ") //Evalúa a "33 // AbC // Falso"
```

Nótese que este ejemplo usa un método de Lista [nativo](#), también llamado *método de tipo*.

Argumentos Opcionales

Cada identificador de argumento puede especificar un valor por defecto en el caso de que no se le suministre un valor al llamar la Función.

Para obtener este comportamiento, simplemente coloca dos puntos y luego el valor.

```
CARGAR sumar con Función(v1: 0, v2: 1)
    DEVOLVER v1 + v2
FIN

ENVIAR sumar(10, 7) //Envía "17"
ENVIAR sumar(5) //Envía "6"
ENVIAR sumar() //Envía "1"
```

No está de más mencionar que los argumentos opcionales van **todos después** de los argumentos requeridos, debido a que realmente no hay forma de “saltarse un argumento” en una expresión de llamado.

Ámbito de Función

El ámbito de una Función no es aquel de un bloque o una sentencia, ni tampoco es global. Este tema está ampliamente cubierto en la [Guía](#).

Nada

Valor primitivo especial

Es un valor que representa “ningún valor”. Es su propio tipo de valor y al mismo tiempo no pertenece a ningún tipo.

El valor Nada puede ser obtenido de:

- Evaluaciones de identificadores que no han sido declarados
- Llamados de Funciones que no devuelven un valor
- La expresión literal Nada

Expresión literal

(Guía). Se expresa como se llama: Nada. **No distingue mayúsculas, minúsculas ni tildes.**

Nada

Conversiones de Nada

- A Texto: Texto Nada = "Nada"
- A Lógico: Lógico Nada = Falso

Ejemplo

El siguiente código enviará “Nada” en el chat. La variable x no existe, así que la expresión evalúa a Nada y se **convierte** a su *representación textual*: "Nada", para poder enviarse.

ENVIAR x

Comprobando Nada en una Variable

Puedes usar la función esNada(x) para comprobar si una expresión – generalmente una variable – vale Nada. Alternativamente, puedes usar el operador es: x es Nada.

Estructuras Estandarizadas

Listado categorizado ordenado

Las *Estructuras Estandarizadas* (también llamadas *Registros Estandarizados* o *Registros Prefabricados*) son plantillas de valores de [Registro](#) usadas para **modelar** de forma estandarizada un concepto, objeto u idea particular. En otras palabras: un molde.

Suelen obtenerse como retorno de ciertas [funciones](#) o [variables](#) nativas.

Estructura Estándar “Miembro”

Un Registro que representa un *miembro* de un servidor de [Discord](#).

Propiedad	Tipo	Descripción
id	Texto	El identificador único del miembro.
avatar	Texto	El enlace de la imagen de perfil del miembro.
nombre	Texto	El nombre del miembro.
mención	Texto	La mención al miembro (<@id>) lista para enviarse.

Estructura Estándar “Canal”

Un Registro que representa un *canal* de [Discord](#).

Propiedad	Tipo	Descripción
id	Texto	El identificador único del canal.
nombre	Texto	El nombre del canal.
mención	Texto	La mención al canal (<#id>) lista para enviarse.

Estructura Estándar “Rol”

Un Registro que representa un *rol* de un servidor de [Discord](#).

Propiedad	Tipo	Descripción
id	Texto	El identificador único del rol.
nombre	Texto	El nombre del rol.
mención	Texto	La mención al rol (<@&id>) lista para enviarse.

Estructura Estándar “Servidor”

Un Registro que representa un *servidor* de **Discord**.

Propiedad	Tipo	Descripción
id	Texto	El identificador único del servidor.
nombre	Texto	El nombre del servidor.
ícono	Texto	El enlace del ícono del servidor.
descripción?	Texto /Nada	La descripción del servidor.
canalSistema?	Registro /Nada	Un Registro representando el canal de sistema del servidor.
cartel?	Texto /Nada	El enlace de la imagen del cartel superior del servidor.
nivel	Texto	El nivel de Boost del servidor.
íImagenInvitación	Texto	El enlace de la imagen de invitación del servidor.
dueño	Registro	Un Registro representando el miembro dueño del servidor.

Estructura Estándar “Marco”

Un Registro que representa un [Marco](#).

Propiedad	Tipo	Descripción
autor?	Registro /Nada	Un Registro del autor del Marco. Por encima de todo.
autor?→nombre	Texto	El nombre del autor del Marco. A la derecha del ícono.
autor?→ícono?	Texto /Nada	El enlace del ícono del autor del Marco. Junto al nombre.
campos	Lista	Una Lista de los campos del Marco. Debajo de la descripción y por encima de la imagen y pie.
<campo>→nombre	Texto	El nombre de un campo. Por encima del valor del mismo.
<campo>→valor	Texto	El valor de un campo. Por debajo del nombre del mismo.
<campo>→alineado	Lógico	Si el campo está alineado (Verdadero) o no (Falso).
color	Texto	El color del borde izquierdo del Marco. Negro por defecto.
descripción	Texto	La descripción del Marco. Debajo del título. "" por defecto.
enlace	Texto /Nada	El enlace principal del Marco.
íImagen?	Registro /Nada	Un enlace de la imagen del Marco. Justo encima del pie.
miniatura?	Texto /Nada	El enlace de la miniatura del Marco. A la derecha.

pie?	Texto /Nada	El pie del Marco. Debajo de los campos y la imagen.
pie?→texto	Texto	El texto del pie del Marco.
pie?→ícono?	Registro /Nada	El enlace del ícono del pie del Marco.
título	Texto	El título del marco. Debajo del autor. "" por defecto.

A tener en cuenta...

A medida que PuréScript evolucione, es probable que los estándares del lenguaje cambien, lo cual incluye estas estructuras.

Variables Nativas

Listado categorizado ordenado

Las siguientes son variables disponibles globalmente. No requieren acceso por medio del operador de flecha (\rightarrow) de ningún tipo. Revisa la tabla:

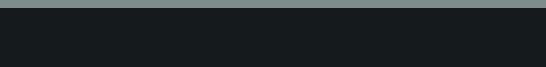
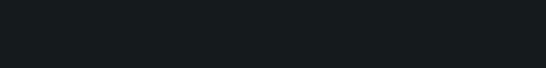
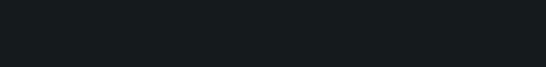
Identificador	Tipo	Descripción
PI	Número	La relación entre la circunferencia de un círculo y su diámetro. Equivalente aproximadamente a 3.141592.
E	Número	La base de los logaritmos naturales y el límite de $\left(1+\frac{1}{n}\right)^n$ cuando n tiende a infinito. Vale aproximadamente 2.71828.
usuario	Registro	El <u>miembro</u> que llama al Tubérculo.
canal	Registro	El <u>canal</u> en el que se llama al Tubérculo.
servidor	Registro	El <u>servidor</u> al que pertenece el Tubérculo.
color(...)	Texto	Puedes ver la lista completa de variables de color aquí .

Colores

Los *valores de color* se representan como [Textos](#) con códigos hexadecimales. Similar a cómo usarías Textos con enlaces para representar imágenes y archivos.

Actualmente, los colores solo sirven para darle color al borde izquierdo de un [Marco](#).

Existen algunos colores predefinidos de forma nativa. Puedes verlos en esta tabla:

Variable o Función	Código	Muestra / Descripción
colorAleatorio()	"#???????"	Un color aleatorio
rgb(rojo, verde, azul)	"#???????"	Un color formado por los valores de rojo , verde y azul indicados (0~255)
hsl(matiz, saturación, luminidad)	"#???????"	Un color según los componentes de matiz (en grados), saturación (0~1) y luminidad (0~1) indicados
hsv(matiz, saturación, brillo)	"#???????"	Un color según los componentes de matiz (en grados), saturación (0~1) y brillo o valor (0~1) indicados
colorAnaranjado	"#E67E22"	
colorAnaranjadoOscuro	"#A84300"	
colorAmarillo	"#FEE75C"	
colorAqua	"#1ABC9C"	
colorAquaOscuro	"#11806A"	
colorAzul	"#3498DB"	
colorAzulOscuro	"#206694"	
colorBlanco	"#FFFFFF"	
colorCasiNegro	"#2C2F33"	"Dark But Not Black" de Discord
colorCeleste	"#1ABC9C"	
colorCelesteOscuro	"#11806A"	
colorDiscord	"#5865F2"	"Blurple" de Discord
colorDorado	"#F1C40F"	
colorDoradoOscuro	"#C27C0E"	
colorFucsia	"#EB459E"	Fucsia oficial de Discord
colorGris	"#95A5A6"	
colorGrisClaro	"#BCC0C0"	
colorGrisNegro	"#7F8C8D"	

colorGrisOscuro	"#979C9F"	
colorGrispura	"#99AAB5"	"Greyle" de Discord
colorMarino	"#34495E"	
colorMarinoOscuro	"#2C3E50"	
colorMarrón	"#A84300"	
colorMorado	"#9B59B6"	
colorMoradoOscuro	"#71368A"	
colorNaranja	"#E67E22"	
colorNaranjaOscuro	"#A84300"	
colorNegro	"#23272A"	"Not Quite Black" de Discord
colorOro	"#F1C40F"	
colorOroOscuro	"#C27C0E"	
colorPúrpura	"#9B59B6"	
colorPúrpuraOscuro	"#71368A"	
colorRojo	"#ED4245"	Rojo oficial de Discord
colorRojoOscuro	"#992D22"	
colorRosaClaro	"#E91E63"	
colorRosaOscuro	"#AD1457"	
colorVerde	"#57F287"	Verde oficial de Discord
colorVerdeOscuro	"#1F8B4C"	
colorVioleta	"#9B59B6"	
colorVioletaOscuro	"#71368A"	

Funciones Nativas

Listado categorizado ordenado

Las *funciones nativas* listadas a continuación son accesibles de forma **global**.

Con tal de dejar claro las diferentes formas de *llamado* de cada Función, se anotará así:

- Se pondrá **siempre** el **identificador** de la Función y paréntesis encerrando los argumentos (haya o no), con tal de hacer alusión a una *expresión de llamado*.
- Se pondrá ***** por detrás de cada **argumento requerido** para diferenciarlos de los **argumentosopcionales**.
- Se pondrá **...** por detrás de un argumento para especificar que este representa un *rango indeterminado* de argumentos en su lugar.
- Se indicarán **todos los tipos** de valor admitibles para **cada argumento**, luego de **:**.
- Se indicará el **tipo** de *valor de retorno u evaluación* de la Función luego del cierre de paréntesis y **→**. Este es el tipo de valor al que evalúa el llamado de la Función.
- Se usará **cualquier** para indicar que un argumento u retorno puede tomar o devolver respectivamente cualquier tipo de valor.

Nótese que tú no debes poner ***** ni **:** ni los **tipos** de argumento u **tipo de retorno**. Estos se anotan de forma meramente ilustrativa, con el fin de informar correctamente del modo de utilización de cada Función.

Existen estas categorías de funciones nativas:

- [Utilidad general](#)
- [Comprobación de tipo](#)
- [Utilidad de Discord](#)
- [Otras comprobaciones](#)

Utilidad General

`aleatorio(a: Número, b: Número) → Número`

Argumento	Tipo	Por defecto	Descripción
a	Número	1	El inicio del rango aleatorio, o el final si no se ingresa b.
b	Número		El final del rango aleatorio.

Sin argumentos, devuelve un Número entre 0 *inclusive* y 1 *exclusive*.

Si se ingresa a, devuelve un Número entre 0 *inclusive* y a *exclusive*.

Si se ingresan a y b, devuelve un Número entre a *inclusive* y b *exclusive*.

Devuelve – Un [Número](#) aleatorio que cumple con los parámetros ingresados.

`colorAleatorio() → Texto`

Devuelve – Un [Texto](#) de código hexadecimal de color aleatorio.

`cos(ángulo*: Número) → Número`

Argumento	Tipo	Por defecto	Descripción
ángulo	Número	Requerido	Un ángulo, en radianes.

Devuelve – Un [Número](#) igual al **coseno** del ángulo especificado (en radianes).

`dado(a: Número, b: Número) → Número`

Argumento	Tipo	Por defecto	Descripción
a	Número		El inicio del rango aleatorio, o el final si no se ingresa b.
b	Número		El final del rango aleatorio.

Tira un dado imaginario y devuelve un Número **entero** aleatorio (sin decimales).

Sin argumentos, es como un dado real, devolviendo algo entre 1 y 6 *inclusive*.

Si se ingresa a, el dado varía entre 0 *inclusive* y a *exclusive*.

Si se ingresan a y b, el dado tira entre a *inclusive* y b *exclusive*.

Devuelve – Un [Número](#) aleatorio que cumple con los parámetros ingresados.

`elegir(valores...*: Cualquier) → Cualquier`

Argumento	Tipo	Por defecto	Descripción
valores...	Cualquier	Requiere al menos 1	Valores de entre los cuáles elegir alguno.

Elige *al azar* entre todos los `valores` y devuelve uno.

Devuelve – Aleatoriamente el valor de alguno de los argumentos ingresados.

`hsl(matiz*: Número, saturación*: Número, luminidad*: Número) → Texto`

Argumento	Tipo	Por defecto	Descripción
matiz	Número	Requerido	Matiz del color, en grados (0 <i>incl.</i> ~ 360 <i>excl.</i>).
saturación	Número	Requerido	Saturación del color. Un factor de 0 a 1 <i>inclusive</i> .
luminidad	Número	Requerido	Luminidad del color. Un factor de 0 a 1 <i>inclusive</i> .

Genera un color en base a los componentes de matiz, saturación y luminidad indicados y lo devuelve.

Devuelve – Un [Texto](#) de código hexadecimal del color formado.

`hsv(matiz*: Número, saturación*: Número, brillo*: Número) → Texto`

Argumento	Tipo	Por defecto	Descripción
matiz	Número	Requerido	Matiz del color, en grados (0 <i>incl.</i> ~ 360 <i>excl.</i>).
saturación	Número	Requerido	Saturación del color. Un factor de 0 a 1 <i>inclusive</i> .
brillo	Número	Requerido	Brillo del color. Un factor de 0 a 1 <i>inclusive</i> .

Genera un color en base a los componentes de matiz, saturación y brillo indicados y lo devuelve.

Devuelve – Un [Texto](#) de código hexadecimal del color formado.

`maximizar(números...*: Número) → Número`

Argumento	Tipo	Por defecto	Descripción
números...	Número	Requiere al menos 1	Números de entre los cuáles elegir el mayor.

Elige de entre todos los números aquel de **mayor valor** y lo devuelve.

Devuelve – Un [Número](#) del valor **máximo** entre argumentos ingresados.

`minimizar(números...*: Número) → Número`

Argumento	Tipo	Por defecto	Descripción
números...	Número	Requiere al menos 1	Números de entre los cuáles elegir el menor.

Elige de entre todos los números aquel de **menor valor** y lo devuelve.

Devuelve – Un [Número](#) del valor **mínimo** entre argumentos ingresados.

`radianes(grados*: Número) → Número`

Argumento	Tipo	Por defecto	Descripción
grados	Número	Requerido	Un ángulo, en grados.

Devuelve – Un [Número](#) resultante de convertir el ángulo de grados a *radianes*.

`raíz(radicando*: Número, grado*: Número) → Número`

Argumento	Tipo	Por defecto	Descripción
radicando	Número	Requerido	El radicando o cuerpo de la expresión radical.
grado	Número	Requerido	El grado u índice de la expresión radical

Calcula la raíz del grado y radicando especificados.

Por ejemplo, para la raíz cuadrada de 9, el radicando sería 9 y el grado sería 2.

Devuelve – Un Número correspondiente a la raíz de la operación radical.

`rgb(rojo*: Número, verde*: Número, azul*: Número) → Texto`

Argumento	Tipo	Por defecto	Descripción
rojo	Número	Requerido	Intensidad del canal rojo del color, de 0 a 255.
verde	Número	Requerido	Intensidad del canal verde del color, de 0 a 255.
azul	Número	Requerido	Intensidad del canal azul del color, de 0 a 255.

Genera un color a partir de la intensidad especificada para los canales de rojo, verde y azul que lo forman y lo devuelve.

Devuelve – Un Texto de código hexadecimal del color formado.

`sen(ángulo*: Número) → Número`

Argumento	Tipo	Por defecto	Descripción
ángulo	Número	Requerido	Un ángulo, en radianes.

Devuelve – Un Número igual al **seno** del ángulo especificado (en radianes).

`tan(ángulo*: Número) → Número`

Argumento	Tipo	Por defecto	Descripción
ángulo	Número	Requerido	Un ángulo, en radianes.

Devuelve – Un Número de la **tangente** del ángulo especificado (en radianes).

Comprobación de Tipo

Estas funciones aceptan cualquier valor e indican si este es del tipo cuestionado o no.

`esNúmero(x*: Cualquier) → Lógico`

Argumento	Tipo	Por defecto	Descripción
x	Cualquier	Requerido	El valor a comprobar.

Verifica si x es de tipo [Número](#).

Devuelve – Verdadero si x es un Número; Falso en cualquier otro caso.

`esTexto(x*: Cualquier) → Lógico`

Argumento	Tipo	Por defecto	Descripción
x	Cualquier	Requerido	El valor a comprobar.

Verifica si x es de tipo [Texto](#).

Devuelve – Verdadero si x es un Texto; Falso en cualquier otro caso.

`esLógico(x*: Cualquier) → Lógico`

Argumento	Tipo	Por defecto	Descripción
x	Cualquier	Requerido	El valor a comprobar.

Verifica si x es de tipo [Lógico](#).

Devuelve – Verdadero si x es un Lógico; Falso en cualquier otro caso.

`esLista(x*: Cualquier) → Lógico`

Argumento	Tipo	Por defecto	Descripción
x	Cualquier	Requerido	El valor a comprobar.

Verifica si x es de tipo [Lista](#).

Devuelve – Verdadero si x es una Lista; Falso en cualquier otro caso.

`esRegistro(x*: Cualquier) → Lógico`

Argumento	Tipo	Por defecto	Descripción
x	Cualquier	Requerido	El valor a comprobar.

Verifica si x es de tipo [Registro](#).

Devuelve – Verdadero si x es un Registro; Falso en cualquier otro caso.

`esMarco(x*: Cualquier) → Lógico`

Argumento	Tipo	Por defecto	Descripción
x	Cualquier	Requerido	El valor a comprobar.

Verifica si x es de tipo [Marco](#).

Devuelve – Verdadero si x es un Marco; Falso en cualquier otro caso.

`esNada(x*: Cualquier) → Lógico`

Argumento	Tipo	Por defecto	Descripción
x	Cualquier	Requerido	El valor a comprobar.

Verifica si x es el valor especial [Nada](#).

Devuelve – Verdadero si x es Nada; Falso en cualquier otro caso.

Utilidad de Discord

buscarMiembro(búsqueda*: Texto) → Registro/Nada

Argumento	Tipo	Por defecto	Descripción
búsqueda	Texto	Requerido	Dato con el cual hacer la búsqueda de miembro.

Busca un miembro de "cercano a lejano" según el Texto de búsqueda ingresado.
La prioridad de búsqueda es la siguiente:

id > tag > nombre de usuario en server actual > apodo en server actual >
nombre de usuario en cualquier server que esté Bot.

Si se encuentra más de un miembro en alguna de las prioridades previas, se determina el escogido según qué tan cerca del primer carácter está el fragmento de Texto encontrado (en relación al nombre encontrado).

Devuelve – Un Registro del [miembro](#) encontrado, o Nada si no se encuentra.

buscarCanal(búsqueda*: Texto) → Registro/Nada

Argumento	Tipo	Por defecto	Descripción
búsqueda	Texto	Requerido	Dato con el cual hacer la búsqueda de canal.

Busca un canal por nombre, mención o id según el Texto de búsqueda ingresado.

Devuelve – Un Registro del [canal](#) encontrado, o Nada si no se encuentra.

buscarRol(búsqueda*: Texto) → Registro/Nada

Argumento	Tipo	Por defecto	Descripción
búsqueda	Texto	Requerido	Dato con el cual hacer la búsqueda de rol.

Busca un rol por nombre, mención o id según el Texto de búsqueda ingresado.

Devuelve – Un Registro del [rol](#) encontrado, o Nada si no se encuentra.

Otras Comprobaciones

`esEnlace(x*: Cualquier) → Lógico`

Argumento	Tipo	Por defecto	Descripción
x	Cualquier	Requerido	El valor a comprobar.

Verifica si `x` es de tipo Texto y si contiene algo que parece un enlace válido, sin importar de qué sea. **No comprueba si es un enlace real.**

Devuelve – Verdadero si `x` es un Texto de enlace; Falso en cualquier otro caso.

`esArchivo(x*: Cualquier) → Lógico`

Argumento	Tipo	Por defecto	Descripción
x	Cualquier	Requerido	El valor a comprobar.

Verifica si `x` es de tipo Texto y si contiene algo que **parece** ser un enlace de archivo válido. **No comprueba si es un enlace a un archivo real.**

Devuelve – Verdadero si `x` parece ser un archivo; Falso en cualquier otro caso.

`esImagen(x*: Cualquier) → Lógico`

Argumento	Tipo	Por defecto	Descripción
x	Cualquier	Requerido	El valor a comprobar.

Verifica si `x` es de tipo Texto y si contiene algo que **parece** ser un enlace de imagen válido. **No comprueba si es un enlace a una imagen real.**

Devuelve – Verdadero si `x` parece ser una imagen; Falso en cualquier otro caso.

Métodos Nativos

Listado categorizado ordenado

Un método es una Función accesible por medio del operador de flecha (\rightarrow) y un nombre.

Si bien solo puedes asignar miembros a [Registros](#), PuréScript define una serie de *propiedades y métodos* que están asociados a **tipos** en lugar de **valores**. Cualquier valor de un determinado tipo puede acceder a las propiedades y los métodos que fueron relacionados nativamente a dicho tipo.

Dado que los métodos nativos están ligados [primitivos](#) y [estructuras](#) de ciertos *tipos*, se les puede llegar a denominar “métodos de tipo”.

Cabe mencionar que los métodos nativos no se consideran **miembros genuinos**.

- Incluso si tienes acceso a 3 métodos de tipo en un Registro y 2 propiedades automáticas, el Registro va a seguir teniendo un tamaño de 0 si no se le carga ningún otro miembro, y convertir el Registro a un Texto solo mostrará un Registro vacío: "`{Rg}`".
- Las propiedades y los métodos nativos no están realmente almacenados en los valores en cuestión, si no que están ligados a tipos de valor.
- Re-asignar propiedades y métodos nativos no tiene efecto alguno, y acceder estos valores y métodos seguirá evaluando a los definidos de forma nativa para el tipo del valor en cuestión.

Con tal de dejar claro las diferentes formas de *llamado* de cada método, se anotará así:

- Se indicará **siempre** el `<tipo de valor>` del que se accede el método, seguido del operador de flecha (\rightarrow) y un pseudo-llamado del método.
- Se pondrá **siempre** el **identificador** del método y paréntesis encerrando los argumentos (haya o no), con tal de hacer alusión a una *expresión de llamado*.
- Se pondrá `*` por detrás de cada **argumento requerido** para diferenciarlos de los **argumentosopcionales**.
- Se pondrá `...` por detrás de un argumento para especificar que este representa un *rango indeterminado* de argumentos en su lugar.
- Se indicarán **todos los tipos** de valor admitibles para **cada argumento**, luego de `:`.
- Se indicará el **tipo** de *valor de retorno u evaluación* del método luego del cierre de paréntesis y \rightarrow . Este es el tipo de valor al que evalúa el llamado del método.
- Se usará `cualquier` para indicar que un argumento u retorno puede tomar o devolver respectivamente cualquier tipo de valor.

Nótese que tú no debes poner `*` ni `:` ni los **tipos** de argumento u **tipo de retorno**. Estos se anotan de forma meramente ilustrativa, con el fin de informar correctamente del modo de utilización de cada método.

Existen estos métodos nativos asociados a los siguientes tipos:

- [Métodos de Número](#)
- [Métodos de Texto](#)
- [Métodos de Lista](#)
- [Métodos de Registro](#)
- [Métodos de Marco](#)

Métodos de Número

Los siguientes métodos pueden ser accedidos desde cualquier [Número](#):

`<número>→absoluto(valor*: Número) → Número`

Siempre devuelve un Número positivo en base al valor de este:

- Si este Número es positivo, devuelve este Número.
- Si este Número es negativo, devuelve el opuesto de este Número.

Devuelve – El valor absoluto de este Número.

`<número>→aEntero() → Número`

Toma la parte entera de este Número y la devuelve. Esto **no** es una operación de redondeo, o sea que la parte decimal es completamente ignorada y no afecta el resultado en lo más mínimo.

El signo no cambia la magnitud obtenida, pero sí se aplica.

Ejemplos: `2.2→aEntero() = 2`, `2.8→aEntero() = 2`, `(-2.8)→aEntero() = -2`

Si quieres aplicar una operación de redondeo, mejor usa [Número→redondear\(\)](#).

Devuelve – Un Número con solo la parte entera de este Número.

<número>→aFijo(precisión*: Número) → Texto

Argumento	Tipo	Por defecto	Descripción
precisión	Número	Requerido	La precisión a mostrar del Número.

Crea una representación de Texto de este número, con una cantidad **fija** de dígitos luego del separador decimal según la **precisión** especificada, y devuelve el Texto. La **precisión** debe ser un número entre 0 y 100 inclusive.

Por ejemplo:

- Una **precisión** de 4 con el Número 3.1415 daría "3.1415"
- Una **precisión** de 3 con el Número 3.5 daría "3.500"
- Una **precisión** de 0 con el Número 2.8 daría "3"
- Una **precisión** de 0 con el Número -2.8 daría "-3"

Devuelve – Una representación textual de este Número con una **precisión fija**.

<número>→aPrecisión(precisión*: Número) → Texto

Argumento	Tipo	Por defecto	Descripción
precisión	Número	Requerido	La precisión máxima a retener del Número.

Crea una representación de Texto de este número, con una cantidad **máxima** de dígitos luego del separador decimal según la **precisión** especificada, y devuelve el Texto. La **precisión** debe ser un número entre 0 y 100 inclusive.

Por ejemplo:

- Una **precisión** de 4 con el Número 3.1415 daría "3.1415"
- Una **precisión** de 3 con el Número 3.5 daría "3.5"
- Una **precisión** de 100 con el Número 42 daría "42"
- Una **precisión** de 0 con el Número 2.8 daría "3"
- Una **precisión** de 0 con el Número -2.8 daría "-3"

Devuelve – Una representación textual de este Número con cierta **precisión**.

```
<número>→aTexto(base: Número) → Texto
```

Argumento	Tipo	Por defecto	Descripción
base	Número	10	La base numérica a usar para la conversión.

Crea una representación textual de este Número utilizando la `base` numérica especificada.

La `base` indica la cantidad de dígitos únicos usados para representar un valor. Por ejemplo: `base-2` tiene los dígitos 0 y 1 (binario), mientras que `base-10` (decimal) tiene de 0 a 9 y `base-16` (hexadecimal) tiene de 0 a F.

Por ejemplo, sabiendo que los Números de PuréScript son **siempre** `base-10`:

- El Número 5 pasado a `base-2` (binario) daría "101"
- El Número 10 pasado a `base-8` (octal) daría "12"
- El Número 10 pasado a `base-16` (hexadecimal) daría "A"
- El Número 42 pasado a `base-16` (hexadecimal) daría "2A"

Si no se pasa una `base`, funciona igual que la conversión: Texto `<número>`.

Devuelve – Un [Texto](#) que representa a este Número en la `base` indicada.

```
<número>→formatear(acortar*: Lógico, mínimoDígitos*: Número) → Texto
```

Argumento	Tipo	Por defecto	Descripción
acortar	Lógico	Requerido	Determina si acortar el número (usando la escala numérica larga) o mostrarlo completo.
mínimoDígitos	Número	Requerido	El mínimo de dígitos en la parte entera.

Crea una representación textual de este Número cuya parte entera tiene el mínimo de dígitos especificado por `mínimoDígitos`. Además, permite `acortar` números grandes de manera dinámica utilizando la [escala numérica larga](#) (o sea que 1234567890 devolvería "1.234 mil millones") o mostrarlos completos.

Devuelve – Un [Texto](#) que representa a este Número formateado acordemente.

```
<número>→limitar(mínimo*: Número, máximo*: Número) → Número
```

Argumento	Tipo	Por defecto	Descripción
mínimo	Número	Requerido	Mínimo del rango del valor.
máximo	Número	Requerido	Máximo del rango del valor.

Restringe este Número dentro del `mínimo` y `máximo` especificados, devolviendo un valor **no menor** que el `mínimo` y **no mayor** que el `máximo`, pero que se acerca *lo más posible* a este valor.

Devuelve – El [Número](#) más cercano a este, entre `mínimo` y `máximo` inclusive.

```
<número>→signo() → Número
```

Devuelve – Un Número (-1, 0 ó 1) indicando el signo de este.

```
<número>→suelo() → Número
```

Devuelve – El Número **entero** más cercano de entre los *menores* que este.

```
<número>→techo() → Número
```

Devuelve – El Número **entero** más cercano de entre los *mayores* que este.

```
<número>→redondear() → Número
```

Devuelve – El Número **entero** más cercano a este.

Métodos de Texto

Los siguientes métodos pueden ser accedidos desde cualquier [Texto](#):

```
<texto>→aLista() → Lista de Textos
```

Genera una Lista a partir de valores de Texto correspondientes a los caracteres de este Texto. El largo de la Lista equivaldrá al largo del Texto.

Devuelve – Una [Lista](#) cuyos elementos son cada carácter de este Texto.

```
<texto>→aMayúsculas() → Texto
```

Devuelve – Un Texto en base a este, con todas las letras en mayúsculas.

```
<texto>→aMinúsculas() → Texto
```

Devuelve – Un Texto en base a este, con todas las letras en minúsculas.

```
<texto>→caracterEn(posición*: Número) → Texto/Nada
```

Argumento	Tipo	Por defecto	Descripción
posición	Número	Requerido	La posición del carácter a encontrar.

Crea un nuevo Texto a partir de uno de los caracteres de este Texto. El carácter seleccionado se decide por la posición especificada. Estando el primer carácter en la posición 0, el segundo en la posición 1, el tercero en 2 y así.

Devuelve – El carácter de este Texto en la posición dada, o [Nada](#) si no existe.

```
<texto>→comienzaCon(subCadena*: Texto) → Lógico
```

Argumento	Tipo	Por defecto	Descripción
subCadena	Texto	Requerido	La cadena de caracteres a usar para comprobar.

Determina si los **primeros** caracteres de este Texto *coinciden exactamente* con la subCadena de caracteres especificada.

Devuelve – Verdadero si este Texto **comienza** con la subCadena especificada.

```
<texto>→contiene(subTexto*: Texto) → Lógico
```

Argumento	Tipo	Por defecto	Descripción
subCadena	Texto	Requerido	La cadena de caracteres a usar para comprobar.

Determina si algún conjunto continuo de caracteres dentro de este Texto *coincide exactamente* con la subCadena de caracteres especificada.

Devuelve – Verdadero si este Texto **incluye** la subCadena especificada.

```
<texto>→cortar(inicio*: Número, fin: Número) → Texto
```

Argumento	Tipo	Por defecto	Descripción
inicio	Número	Requerido	El inicio del sub-texto a recortar. Inclusive.
fin	Número	...→largo	El fin del sub-texto a recortar. Exclusive.

Genera un nuevo Texto a partir de un recorte de este Texto desde el carácter de `inicio` hasta el carácter de `fin`, sin incluir el carácter exactamente en `fin`. La posición se cuenta desde `0` para el primer carácter.

Existen estas excepciones:

- Si este Texto está vacío, siempre se crea un Texto vacío.
- Si `inicio` excede el largo de este Texto, se crea un Texto vacío.
- Si `fin` excede el largo de este Texto, no se recorta el final.
- Si alguna de 2 posiciones es un Número negativo, se la cuenta desde el último carácter hacia atrás (en lugar de desde el primero hacia delante).
- Si `fin` acaba siendo menor que `inicio` luego de tener en cuenta la conversión de posiciones de Números negativos, se crea un Texto vacío.

Devuelve – Un nuevo Texto en base a un recorte de `inicio` a `fin` de este.

```
<texto>→normalizar() → Texto
```

Crea una versión de este Texto sin caracteres en blanco al inicio ni al final. Los caracteres en blanco entre medio de otros caracteres no son eliminados.

Devuelve – Un Texto igual a este pero sin caracteres en blanco alrededor.

```
<texto>→partir(separador*: Texto) → Lista de Textos
```

Argumento	Tipo	Por defecto	Descripción
separador	Texto	Requerido	El Texto a buscar para partir en elementos.

Divide este Texto en base al `separador` especificado, resultando en una nueva Lista ordenada de las sub-cadenas de caracteres obtenidas luego de dividir.

Devuelve – Una [Lista](#) de los contenidos de este Texto separados acordemente.

<texto>→posiciónDe(búsqueda*: Texto) → Número

Argumento	Tipo	Por defecto	Descripción
búsqueda	Texto	Requerido	El Texto a encontrar dentro de este.

Se busca el Texto de búsqueda especificado dentro de este Texto, de inicio a fin o izquierda a derecha, hasta encontrar una *coincidencia exacta*.

Si se encuentra la secuencia, se devuelve la posición del primer carácter de la coincidencia. Dicha posición se cuenta desde 0 y es en relación a este Texto.

Devuelve – La posición del **primer** carácter de la **primer** coincidencia de búsqueda encontrada en relación a este Texto, ó -1 si no se encuentra nada.

<texto>→reemplazar(ocurrencia*: Texto, reemplazo*: Texto) → Texto

Argumento	Tipo	Por defecto	Descripción
ocurrencia	Texto	Requerido	El sub-texto a detectar para reemplazar.
reemplazo	Texto	Requerido	El reemplazo que se aplicará a las ocurrencias.

Crea un Texto basado en sustituir todas las ocurrencias encontradas en este Texto por el reemplazo especificado.

Devuelve – Un nuevo Texto como este pero con los reemplazos aplicados.

<texto>→repetido(veces*: Número) → Texto

Argumento	Tipo	Por defecto	Descripción
veces	Número	0	La cantidad de veces a repetir el Texto.

Genera un nuevo Texto a base de repetir este Texto la cantidad de veces indicada. El Número ingresado debe ser 0 ó más, y no debería ser muy grande.

Si el Número es 0, se genera un Texto vacío. Si el Número es 1, se crea un Texto idéntico a este. Si el Número es 2, se crea un Texto que tiene 2 veces al original, y así se extrae para el resto de valores.

Devuelve – Un Texto en base a repetir este la cantidad de veces indicada.

```
<texto>→terminaCon(subCadena*: Texto) → Lógico
```

Argumento	Tipo	Por defecto	Descripción
subCadena	Texto	Requerido	La cadena de caracteres a usar para comprobar.

Determina si los **últimos** caracteres de este Texto *coinciden exactamente* con la subCadena de caracteres especificada.

Devuelve – Verdadero si este Texto **termina** con la subCadena especificada.

```
<texto>→últimaPosiciónDe(búsqueda*: Texto) → Número
```

Argumento	Tipo	Por defecto	Descripción
búsqueda	Texto	Requerido	El Texto a encontrar dentro de este.

Mismo comportamiento que `Texto→posiciónDe(búsqueda)`, pero se devuelve en base a la **última** coincidencia encontrada en lugar de la primera.

Devuelve – La posición del **primer** carácter de la **última** coincidencia de búsqueda encontrada en relación a este Texto, ó `-1` si no se encuentra nada.

Métodos de Lista

Los siguientes métodos pueden ser accedidos desde cualquier [Lista](#):

```
<lista>→aInvertida() → Lista
```

Genera una Lista a partir de esta, con todos los elementos ordenados al revés.

Devuelve – Una Lista con los elementos de la original ordenados al revés.

```
<lista>→alguno(predicado*: Función) → Lógico
```

Argumento	Tipo	Por defecto	Descripción
predicado	Función	Requerido	Una Función de predicado.

Determina si en esta Lista existe algún elemento tal que el `predicado` devuelva un valor convertible a `Verdadero` al pasarle dicho elemento como argumento.

Devuelve – [Verdadero](#) si algún elemento de esta Lista satisface el `predicado`.

```
<lista>→aOrdenada(criterio: Función) → Lista
```

Argumento	Tipo	Por defecto	Descripción
criterio	Función		Una Función de comparación.

Genera una Lista con todos los elementos de esta Lista, reordenados en base al `criterio` especificado.

Si no se suministra una Función de `criterio`, los elementos se ordenan de menor a mayor en base al criterio de ordenamiento del tipo de cada elemento.

La Función de `criterio` debería tomar dos argumentos y devolver un Número:

- Si fuera menor que 0, significaría que el primer valor precede al segundo.
- Si fuera mayor que 0, significaría que el primer valor excede al segundo.
- Si fuera 0, significaría que ambos argumentos “valen lo mismo”.

Devuelve – Una [Lista](#) con los elementos de esta ordenados según el `criterio`.

```
<lista>→aRegistro() → Registro
```

Crea un Registro a partir de los elementos de esta Lista. El tamaño del Registro equivaldrá al largo de esta Lista.

Las claves y valores de los miembros del registro corresponderán al índice y valor cada elemento, respectivamente.

Devuelve – Una representación de Registro de esta Lista.

```
<lista>→contiene(valor*: Cualquier) → Lógico
```

Argumento	Tipo	Por defecto	Descripción
valor	Cualquier	Requerido	El valor a comprobar entre los elementos.

Determina si esta Lista incluye algún elemento con el `valor` especificado.

Devuelve – Verdadero si esta Lista incluye el valor especificado.

```
<lista>→cortar(inicio*: Número, fin: Número) → Lista
```

Argumento	Tipo	Por defecto	Descripción
inicio	Número	Requerido	La primer posición a incluir. Inclusive.
fin	Número	→largo	La última posición a incluir. Exclusive.

Genera una nueva Lista a partir de un recorte de esta Lista de `inicio` a `fin`, comenzando desde el elemento en la posición de `inicio` y sin incluir desde el elemento posicionado exactamente en `fin`.

Hay algunas excepciones:

- Si esta Lista está vacía, se genera una Lista vacía.
- Si `inicio` excede el largo de esta Lista, se genera una Lista vacía.
- Si `fin` excede el largo de esta Lista, no se recorta el final.
- Si alguna de 2 posiciones es un Número negativo, esta se contará desde el último elemento hacia atrás.
- Si `fin` acaba siendo menor que `inicio` luego de tener en cuenta la conversión de posiciones de Números negativos, se crea una Lista vacía.

Devuelve – Una Lista como esta pero recortada de `inicio` a `fin`.

```
<lista>→encontrar(predicado*: Función) → Cualquier
```

Argumento	Tipo	Por defecto	Descripción
predicado	Función	Requerido	Una Función de predicado.

Devuelve el **primer** elemento de esta Lista para el cual el `predicado` devuelve un valor convertible a Verdadero al pasarle dicho elemento como argumento.

Si no se encuentra ningún elemento que cumpla el predicado, se devuelve Nada.

Devuelve – El **primer** elemento de esta Lista que satisface el `predicado`, o Nada.

```
<lista>→encontrarId(predicado*: Función) → Número
```

Argumento	Tipo	Por defecto	Descripción
predicado	Función	Requerido	Una Función de predicado.

Devuelve la *posición* del **primer** elemento de esta Lista para el cual el `predicado` devuelve un valor convertible a Verdadero al pasárselo como argumento.

Si no se encuentra ningún elemento que cumpla el predicado, se devuelve -1.

Devuelve – El índice del **primer** elemento de esta Lista que satisface el `predicado`, ó -1 si ninguno lo hace.

```
<lista>→encontrarÚltimo(predicado*: Función) → Cualquier
```

Argumento	Tipo	Por defecto	Descripción
predicado	Función	Requerido	Una Función de predicado.

Devuelve el **último** elemento de esta Lista para el cual el `predicado` devuelve un valor convertible a Verdadero al pasarle dicho elemento como argumento.

Si no se encuentra ningún elemento que cumpla el predicado, se devuelve Nada.

Devuelve – El **último** elemento de esta Lista que satisface el `predicado`, o Nada.

```
<lista>→encontrarÚltimoId(predicado*: Función) → Número
```

Argumento	Tipo	Por defecto	Descripción
predicado	Función	Requerido	Una Función de predicado.

Devuelve la posición del **último** elemento de esta Lista para el cual el **predicado** devuelve un valor convertible a Verdadero al pasárselo como argumento.

Si no se encuentra ningún elemento que cumpla el **predicado**, se devuelve -1.

Devuelve – El índice del **último** elemento de esta Lista que satisface el **predicado**, ó -1 si ninguno lo hace.

```
<lista>→filtrar(filtro*: Función) → Lista
```

Argumento	Tipo	Por defecto	Descripción
filtro	Función	Requerido	Una Función que determina si conservar un determinado elemento o no.

Genera una nueva Lista con todos los elementos de esta Lista, excepto aquellos que al ser pasados como argumento a la Función de **filtro** no devuelven un valor convertible a Verdadero.

Devuelve – Una Lista con solo los elementos originales que pasan el **filtro**.

```
<lista>→invertir() → Nada
```

Espeja el orden los elementos de esta Lista. Resulta en que el índice 0 pase a ser el último índice y vice-versa, el índice 1 el ante-último y vice-versa, y así.

```
<lista>→mapear(mapeo*: Función) → Lista
```

Argumento	Tipo	Por defecto	Descripción
mapear	Función	Requerido	Una Función que determina cómo transformar o mapear cada elemento a otra Lista.

Genera una nueva Lista con todos los elementos de esta Lista pero transformados según lo especifique la Función de `mapear` dada.

La Función de mapeo debería tomar un argumento (el elemento) y devolver un valor de cualquier tipo en base al mismo (la transformación del elemento).

Devuelve – Una Lista con los elementos transformados según el `mapear` ofrecido.

```
<lista>→ordenar(criterio: Función) → Nada
```

Argumento	Tipo	Por defecto	Descripción
criterio	Función		Una Función de comparación.

Ordena los elementos de esta Lista en base al `criterio` especificado.

Si no se suministra una Función de `criterio`, los elementos se ordenan de menor a mayor en base al criterio de ordenamiento del tipo de cada elemento.

```
<lista>→paraCada(procedimiento*: Función) → Nada
```

Argumento	Tipo	Por defecto	Descripción
procedimiento	Función	Requerido	Una Función que hace algo con un elemento.

Ejecuta la Función de `procedimiento` facilitada usando cada elemento de esta Lista como argumento, uno por uno y en orden de indizado.

```
<lista>→robar(posición*: Número) → Cualquier
```

Argumento	Tipo	Por defecto	Descripción
posición	Número	Requerido	El Texto a colocar entre los elementos unidos.

Remueve el elemento en la posición indicada de esta Lista.

Devuelve – El elemento eliminado de esta Lista.

```
<lista>→robarPrimero() → Cualquier
```

Remueve el primer elemento de esta Lista.

Devuelve – El elemento eliminado de esta Lista.

```
<lista>→robarÚltimo() → Cualquier
```

Remueve el último elemento de esta Lista.

Devuelve – El elemento eliminado de esta Lista.

```
<lista>→todos(predicado*: Función) → Lógico
```

Argumento	Tipo	Por defecto	Descripción
predicado	Función	Requerido	Una Función de predicado.

Determina si el predicado devuelve un valor convertible a Verdadero para todos los elementos de esta Lista al pasárselos individualmente como argumentos.

Devuelve – Verdadero si todos los elemento de esta Lista cumplen el predicado.

```
<lista>→unir(separador: Texto) → Texto
```

Argumento	Tipo	Por defecto	Descripción
separador	Texto	" , "	El Texto a colocar entre los elementos unidos.

Genera un nuevo Texto en base a las *representaciones textuales* de todos los elementos de esta Lista, en orden y separadas por el separador especificado.

Si esta Lista está vacía, se genera un Texto vacío.

Puedes pasar "" para imitar el comportamiento de Texto <lista>.

Devuelve – Un Texto de todos los elementos de esta Lista unidos acordemente.

```
<lista>→último() → Cualquier
```

Devuelve – El último elemento de esta Lista.

```
<lista>→vacía() → Lógico
```

Determina si esta Lista no contiene ningún elemento.

Devuelve – Verdadero si esta Lista está vacía.

Métodos de Registro

Los siguientes métodos pueden ser accedidos desde cualquier [Registro](#):

`<registro>→claves() → Lista de Textos`

Genera una Lista con solo las claves de todos los miembros de este Registro.

El orden de las claves **no** está determinado.

Devuelve – Una [Lista](#) con las claves de las entradas de este Registro.

`<registro>→contiene(clave*: Texto) → Lógico`

Argumento	Tipo	Por defecto	Descripción
clave	Texto	Requerido	El nombre de la clave a verificar.

Devuelve – [Verdadero](#) si este Registro tiene la `clave` especificada.

`<registro>→entradas() → Lista de Listas`

Genera una Lista con Listas que representan las entradas de este Registro.

La Lista que contiene otras Listas tiene un largo equivalente al tamaño del Registro, dado que contiene las entradas en forma de Lista.

Cada Lista interna tiene un largo exacto de 2 y representa una entrada. La posición 0 contiene su clave y la posición 1 contiene su valor.

Devuelve – Una [Lista](#) con las entradas de este Registro en forma de Listas de 2.

```
<registro>→filtrar(filtro*: Función) → Registro
```

Argumento	Tipo	Por defecto	Descripción
filtro	Función	Requerido	Una Función que determina si conservar una determinada entrada o no.

Genera un nuevo Registro con todas las entradas de este Registro, excepto aquellas que al ser pasadas como argumentos de clave-valor a la Función de filtro no devuelven un valor convertible a Verdadero.

Por cada entrada, se pasa un Texto de la clave como primer argumento y el valor asociado como segundo argumento. Por ende, la Función de filtro debería tomar hasta 2 argumentos y determinar si conservar la entrada o no.

Devuelve – Un Registro con solo las entradas originales que pasan el filtro.

```
<registro>→paraCada(procedimiento*: Función) → Nada
```

Argumento	Tipo	Por defecto	Descripción
procedimiento	Función	Requerido	Una Función que hace algo con una entrada.

Ejecuta la Función de procedimiento facilitada usando cada entrada de este Registro como argumento, una por una.

Por cada entrada, se pasa un Texto de la clave como primer argumento y el valor asociado como segundo argumento. Por ende, la Función de procedimiento debería tomar hasta 2 argumentos.

```
<registro>→quitar(clave*: Texto) → Lógico
```

Argumento	Tipo	Por defecto	Descripción
clave	Texto	Requerido	El nombre de la clave a quitar.

Remueve de este Registro la entrada con la clave especificada.

Devuelve – Verdadero si la clave existía y fue removida, o Falso de lo contrario.

```
<registro>→vacío() → Lógico
```

Determina si este Registro no contiene ninguna entrada.

Devuelve – Verdadero si este Registro está vacío.

```
<registro>→valores() → Lista
```

Genera una Lista con solo los valores de todos los miembros de este Registro.

El orden de los valores **no** está determinado.

Devuelve – Una Lista con los valores de las entradas de este Registro.

Métodos de Marco

Así se ve un Marco con 2 campos y todas sus propiedades asignadas:



Los siguientes métodos pueden ser accedidos desde cualquier [Marco](#):

```
<marco>→agregarCampo(nombre*: Texto, valor*: Texto, alineado: Lógico) → Marco
```

Argumento	Tipo	Por defecto	Descripción
nombre	Texto	Requerido	El nombre del campo a añadir. Sobre el valor.
valor	Texto	Requerido	El valor del campo a añadir. Bajo el nombre.
alineado	Lógico	Falso	Si va en la misma columna que otros campos.

Añade un campo a este Marco con el `nombre` y el `valor` especificados. No pueden ser Textos vacíos.

Pueden haber entre **1** y **3** campos `alineados` en la misma columna, y pueden haber hasta **25** campos en un mismo Marco.

Devuelve – Este Marco.

Nombre de Campo 1

Valor de Campo 1

Nombre de Campo 2

Valor de Campo 2

Alineado 1 Alineado 2 Alineado 3

Valor Valor Valor

```
<marco>→aRegistro() → Registro
```

Devuelve – Una [representación estandarizada de Registro](#) de este Marco.

```
<marco>→asignarAutor(nombre*: Texto, ícono: Texto) → Marco
```

Argumento	Tipo	Por defecto	Descripción
nombre	Texto	Requerido	El nombre del autor a mostrar.
ícono	Texto		El enlace para el ícono del autor a mostrar.

Asigna un autor a este Marco con el `nombre` y el `ícono` especificados. No se puede pasar un nombre vacío y el enlace del ícono debe ser válido.



Si el Marco ya tenía un autor, se lo reemplaza.

Devuelve – Este Marco.

A tener en cuenta...

El "autor" aquí hace referencia a *qué se muestra* como autor en el Marco. No debe confundirse con el autor del Tubérculo (el Programador) ni mucho menos con quien llama al Tubérculo (el [usuario](#)).

```
<marco>→asignarColor(color*: Texto) → Marco
```

Argumento	Tipo	Por defecto	Descripción
color	Texto	Requerido	El color a asignar.

Reemplaza el `color` de este Marco por el especificado. No se puede pasar un Texto vacío como color. Puedes ingresar un código hexadecimal o usar un [color predefinido](#).

← color

Por defecto, el color de un Marco es el código hexadecimal "#000000" (negro).

Devuelve – Este Marco.

```
<marco>→asignarDescripción(descripción*: Texto) → Marco
```

Argumento	Tipo	Por defecto	Descripción
descripción	Texto	Requerido	La descripción a mostrar.

Asigna una descripción a este Marco. No se puede pasar una descripción vacía.

Descripción

Si el Marco ya tenía una descripción, se la reemplaza.

Devuelve – Este Marco.

```
<marco>→asignarEnlace(enlace*: Texto) → Marco
```

Argumento	Tipo	Por defecto	Descripción
enlace	Texto	Requerido	El enlace a colocar.

Asigna un enlace a este Marco. El mismo puede ser clickeado por medio del **título** del marco. El Texto del enlace debe ser un enlace válido.

Si el Marco ya tenía una imagen, se la reemplaza.

Devuelve – Este Marco.

```
<marco>→asignarImagen(imagen*: Texto) → Marco
```

Argumento	Tipo	Por defecto	Descripción
imagen	Texto	Requerido	El enlace de imagen a mostrar.

Asigna una imagen a este Marco. El Texto del enlace de la imagen debe ser un enlace válido.

Título

Descripción

Si el Marco ya tenía una imagen, se la reemplaza.



Devuelve – Este Marco.

```
<marco>→asignarMiniatura(miniatura*: Texto) → Marco
```

Argumento	Tipo	Por defecto	Descripción
miniatura	Texto	Requerido	El enlace de imagen de miniatura a mostrar.

Asigna una imagen de miniatura a este Marco. El Texto del enlace de la miniatura debe ser un enlace válido.
Si el Marco ya tenía una miniatura, se la reemplaza.

Devuelve – Este Marco.



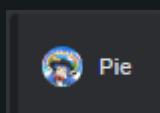
```
<marco>→asignarPie(texto*: Texto, ícono: Texto) → Marco
```

Argumento	Tipo	Por defecto	Descripción
texto	Texto	Requerido	El texto del pie a mostrar.
ícono	Texto		El enlace del ícono del pie a mostrar.

Asigna un pie a este Marco con el texto y el ícono especificados. No se puede pasar un texto vacío y el enlace de la imagen debe ser válido.

Si el Marco ya tenía un pie, se lo reemplaza.

Devuelve – Este Marco.



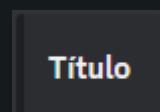
```
<marco>→asignarTítulo(título*: Texto) → Marco
```

Argumento	Tipo	Por defecto	Descripción
título	Texto	Requerido	El título a mostrar.

Asigna un título al marco. No se puede pasar un título vacío.

Si el Marco ya tenía un título, se lo reemplaza.

Devuelve – Este Marco.



Lista de Operadores

Listado categorizado ordenado

Los operadores son el componente principal de una operación, ya que definen qué se hará con el/los operandos para llegar a un resultado, el tipo de operandos que admite la operación, y el tipo de resultado u evaluación de la operación.

Puedes aprender más sobre operadores en la [Guía](#).

Operadores Aritméticos

/ [Guía - Componentes Fundamentales - Tipos de Operador](#)

Los siguientes operadores se usan en lo que se suele referir como operaciones matemáticas. Aceptan [Números](#) o cualquier dato que pueda convertirse a un Número (excepto [Textos](#), ya que estos indican una [operación de concatenación](#)):

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... + ...	Convertible a Número (Excepto Texto)	Convertible a Número (Excepto Texto)	Número

Suma ambos sumandos, evaluando al total de la [adición](#).

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... - ...	Convertible a Número	Convertible a Número	Número

Resta el sustraendo derecho al minuendo izquierdo, evaluando a la diferencia de la [resta](#).

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... * ...	Convertible a Número	Convertible a Número	Número

Multiplica ambos factores, evaluando al producto de la [multiplicación](#).

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... / ...	Convertible a Número	Convertible a Número	Número

Divide el dividendo izquierdo por el divisor derecho, evaluando al cociente de la [división](#).

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... % ...	Convertible a Número	Convertible a Número	Número

Divide el dividendo izquierdo por el divisor derecho, evaluando al [resto](#) de la [división](#).

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... ^ ** ...	Convertible a Número	Convertible a Número	Número

Potencia la base izquierda por el exponente derecho, evaluando a la [potenciación](#).

Tipo	Operador	Argumento	Evaluación
Unario	+ ...	Convertible a Número	Número

Evalúa a la representación numérica del operando. Es una forma rápida de representar valores como Números.

Tipo	Operador	Argumento	Evaluación
Unario	- ...	Convertible a Número	Número

Evalúa a la representación numérica invertida del operando.

Operador de Concatenación

/ [Guía - Componentes Fundamentales - Tipos de Operador](#)

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... + ...	Convertible a Texto	Convertible a Texto	Texto

Acopla el Texto del operando derecho al final del Texto del operando izquierdo, efectivamente combinando los caracteres de ambos Textos.

Operador de Dos Puntos ":"

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... : ...	Identificador	Cualquier	N/A

Se usa en expresiones literales de Registro para asignar entradas a estos. El operando izquierdo representa la clave de una entrada, mientras que el operando derecho representa su respectivo valor.

Operador de Flecha "->"

/ [Guía - Componentes Fundamentales - Expresiones de Acceso a Miembro](#)

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... → ...	Cualquier (Excepto Nada)	Identificador o Número	Cualquier

Se usa en expresiones de flecha para acceder a un valor, variable o método miembro (conocido como “propiedad”) de un determinado valor (conocido como “contenedor”).

El operando izquierdo representa el contenedor, mientras que el operando derecho representa el miembro o la propiedad que se accede. La operación evalúa al valor indicado dentro del contenedor referenciado.

Se pueden concatenar múltiples operaciones de flecha para acceder progresivamente a los miembros dentro de los miembros del contenedor principal.

La propiedad accedida es aquella que se escribe *literalmente*, en lugar de una evaluación.

Operador de Acceso Computado “->(...)”

/ [Guía - Componentes Fundamentales - Expresiones de Acceso a Miembro](#)

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	<code>... →(...)</code>	Cualquier (Excepto Nada)	Cualquier	Cualquier

Contrario al operador de flecha tradicional, este operador permite poner una expresión entre paréntesis que actuará como operando derecho y será evaluada para determinar qué identificador acceder en el contenedor del operando izquierdo.

El valor resultante de la expresión entre los paréntesis es lo que será utilizado como clave o número de acceso a fin de cuentas.

Operadores Conectores Lógicos

/ [Guía - Persistencia de Datos - Carga de Datos y Declaración Condicional](#)

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	<code>... y ...</code> <code>... & ...</code>	Cualquier	Cualquier	Cualquier

Evalúa a uno de los dos operandos dependiendo de la representación lógica del operando izquierdo. Incluye [evaluación de cortocircuito](#). Formalmente, funciona así:

- Si el operando izquierdo es Falso, se evalúa al operando izquierdo.
- Si el operando izquierdo es Verdadero, se evalúa al operando derecho.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	<code>... o ...</code> <code>... ...</code>	Cualquier	Cualquier	Cualquier

Evalúa a uno de los dos operandos dependiendo de la representación lógica del operando izquierdo. Incluye [evaluación de cortocircuito](#). Formalmente, funciona así:

- Si el operando izquierdo evalúa a Falso, se evalúa al operando izquierdo.
- Si el operando izquierdo evalúa a Verdadero, se evalúa al operando derecho.

Operador de Negación

Tipo	Operador	Argumento	Evaluación
Unario	no ... ! ...	Convertible a Lógico	Lógico

Invierte el valor Lógico del argumento. Evalúa al resultado de la inversión.

Según la representación lógica evaluada del argumento:

- Si fuese Verdadero, la operación evaluaría a Falso en su lugar.
- Si fuese Falso, la operación evaluaría a Verdadero en su lugar.

Operadores Relacionales

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... es == ...	Cualquier	Cualquier	Lógico

Si ambos operandos valen exactamente lo mismo y son del mismo tipo, se evalúa a Verdadero. Si no coinciden en tipo, valor, o ambos, se evalúa a Falso.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... no es != ...	Cualquier	Cualquier	Lógico

Considera el mismo criterio que es e invierte el resultado.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... parece ...	Cualquier	Cualquier	Lógico

Si los operandos evalúan a valores similares entre sí, se evalúa a Verdadero. Si los valores no son similares, se evalúa a Falso.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... no parece ...	Cualquier	Cualquier	Lógico

Considera el mismo criterio que parece e invierte el resultado.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... precede < ...	Cualquier	Cualquier	Lógico

Si el operando izquierdo es de menor orden que el derecho, evalúa a Verdadero. Alternativamente, evalúa a Falso.

Si los tipos de los valores no coinciden, ocurre un arbitraje que dice que el primer valor es de menor orden que el segundo.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... no precede >= ...	Cualquier	Cualquier	Lógico

Considera el mismo criterio que precede e invierte el resultado. También puede verse como comprobar si el operando izquierdo es o excede el operando derecho.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... excede > ...	Cualquier	Cualquier	Lógico

Si el operando izquierdo es de un mayor orden que el derecho, evalúa a Verdadero. Alternativamente, evalúa a Falso.

Si los tipos de los valores no coinciden, ocurre un arbitraje que dice que el primer valor es de menor orden que el segundo.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... no excede <= ...	Cualquier	Cualquier	Lógico

Considera el mismo criterio que excede e invierte el resultado. También puede verse como comprobar si el operando izquierdo es o precede el operando derecho.

Operador de Llamado

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... (...)	Función	*	Cualquier
Llama al valor de Función evaluado del operando izquierdo, con los valores evaluados de las expresiones separadas por comas entre los paréntesis a la derecha como argumentos para la ejecución del llamado.				

La operación de llamado evalúa al valor devuelto (o valor de retorno) de la Función llamada.

Operador Lambda

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... => ...	Identificador	Cualquier	Función
Binario	(...) => ...	Identificador *	Cualquier	Función

Crea una Función con los argumentos indicados por el operando izquierdo y la expresión de retorno del operando derecho.

A la Función no se le expresa un cuerpo explícito, y en su lugar se ofrece una expresión de retorno que indica qué devolverá inmediatamente la Función creada cuando sea llamada.

Si el operando izquierdo no está envuelto en paréntesis, entonces debe ser un solo identificador. Si está envuelto en paréntesis, será una secuencia de identificadores separados por comas. Estos son los argumentos de la Función.

Operadores de Agrupación "()"

Tipo	Operador	Argumento	Evaluación
Encapsulación	(...)	Cualquier	Cualquier

Encapsula expresiones y les da [mayor precedencia](#) sobre toda operación.

Operador de Coma ","

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	... , ...	Cualquier	Cualquier	Cualquier

Se usa en [literales de Lista](#), [literales de Registro](#), expresiones de [argumentos de Función](#) y [expresiones de secuencia](#) como un medio para separar expresiones en un formato de listado.

En el caso de expresiones de secuencia, su evaluación causa que se evalúen de izquierda a derecha todas las expresiones separadas por comas, y el resultado de esta expresión sería el resultado de la *última* expresión evaluada.

Tabla de Precedencias

Listado categorizado ordenado

Las [operaciones](#) binarias son construidas a partir de 2 operandos y 1 operador (ó 1 operando y 1 operador en el caso de expresiones unarias).

En casos aislados de operaciones simples se puede simplemente evaluar la operación y conseguir un valor, pero en el caso de múltiples operaciones (conocidas en conjunto como operaciones complejas) se debe seguir un orden de evaluación para llegar a un solo resultado final. Aquí entra en juego la **precedencia de operadores**.

Básicamente, las operaciones siguen un orden de evaluación determinado por el valor de precedencia de su operador. Una mayor precedencia indica que la operación se evaluará antes que una con menor precedencia.

Para ilustrar esto, podemos tomar de ejemplo el orden de precedencia matemático, en el cuál la multiplicación y división tienen una mayor precedencia que la suma y la resta:

```
2 + 3 * 4
```

Si no consideráramos la precedencia, esto se podría evaluar de 2 formas:

```
(2 + 3) * 4 = 5 * 4 = 20  
2 + (3 * 4) = 2 + 12 = 14
```

Sin embargo, el orden de precedencia indica que primero se debe realizar la multiplicación, así que si no colocamos paréntesis esto evalúa a 14.

Así mismo, cuando las precedencias *coinciden*, se recurre a la **asociatividad**.

La asociatividad indica si operadores de igual precedencia deben evaluarse:

- **Asociados a la izquierda** — *De izquierda a derecha*. Las operaciones más a la izquierda se evalúan primero.
- **Asociados a la derecha** — *De derecha a izquierda*. Las operaciones más a la derecha se evalúan primero.

Las sumas y restas tienen la misma precedencia y asociatividad a la izquierda. Esto significa que las de más a la izquierda pasan a ser operandos de los operadores de la derecha. Pasa lo mismo con multiplicaciones y divisiones. Por ejemplo:

$$\begin{aligned} 2 + 4 - 1 + 3 &= 8 \\ 3 * 4 / 2 * 5 &= 30 \end{aligned}$$

Podemos ilustrar el orden de evaluación con paréntesis de la siguiente forma:

$$\begin{aligned} ((2 + 4) - 1) + 3 &= 8 \\ ((3 * 4) / 2) * 5 &= 30 \end{aligned}$$

Con esto en cuenta, sabemos que primero se comprueba la precedencia y, si las precedencias coinciden, se comprueba también la asociatividad.

Veamos un ejemplo que combina ambos casos:

$$5 + 4 * 3 ^ 2 / 2 - 1 = 22$$

Si ilustramos esto con paréntesis, se vería así:

$$\begin{aligned} (5 + (4 * (3 ^ 2) / 2)) - 1 &= \\ (5 + ((4 * 9) / 2)) - 1 &= \\ (5 + (36 / 2)) - 1 &= \\ (5 + 18) - 1 &= \\ 23 - 1 &= 22 \end{aligned}$$

Pero esto no aplica *solo* a operaciones matemáticas, sino que aplica a **todos** los operadores de PuréScript. Refiérete a la tabla debajo:

Precedencia	Asociatividad	Operadores	Descripción
12	N/A	(...) ... → ...	Agrupación Acceso a Miembros
11	A la izquierda	... →(...) ... (...) no ... ! ... + ... - ...	Acceso Computado a Miembros Llamado a Función Negación lógica
10	N/A	+ ... - ...	Suma Unaria Resta Unaria
9	A la derecha	... ^ ** * ...	Potenciación Multiplicación
8	A la izquierda	... / % ...	División Módulo
7	A la izquierda	... + - precede < no precede ≥ excede > no excede ≤ es = no es ≠ parece no parece y & o ⇒ , ...	Adición/Concatenación Sustracción Comparación Igualdad Desigualdad Similaridad De-similaridad Conjunción lógica Disyunción lógica Lambda Coma
6	A la izquierda		
5	A la izquierda		
4	A la izquierda		
3	A la izquierda		
2	A la derecha		
1	A la izquierda		

Glosario

Este capítulo cubre conceptos comunes de programación que te pueden servir para una multitud de otros lenguajes, o términos comúnmente utilizados usados en PuréScript.

Es mejor acompañado con los capítulos de [Guía](#) y [Referencia](#).

Índice de Capítulo

Glosario	245
Token	246
Identificador	247
Primitivo/a	248
Estructura	249
Usuario	250



Token

Un token léxico es algún texto de código fuente con un significado asignado. Suelen estructurarse como un par de «nombre de token» y «valor de token» opcional. El nombre del token es una categoría de unidad léxica. Por ejemplo:

- **Identificador** — nombres que elige el programador
- **Palabra clave** — nombres reservados de PuréScript
- **Operador** — símbolos que operan sobre argumentos y producen resultados
- **Literal** — Números, Texto, etc., escritos literalmente.

(*Información extraída de [Wikipedia](#)*).

¿Te perdiste? Vuelve a:

- [Guía – Gramática – Análisis Léxico e Interpretación](#)

Identificador

Un *identificador* es una secuencia de caracteres que identifica una variable, Función o propiedad. Son las únicas palabras en PuréScript que se distinguen con **mayúsculas, minúsculas y tildes**, además de aquellas palabras dentro de un literal de Texto.

A diferencia de los literales de Texto, que son datos de Texto, los identificadores son símbolos en el código que sirven para referenciar partes con nombre en el programa.

¿Te perdiste? Vuelve a:

- [Guía – Gramática – Sentencias](#)
- [Guía – Declaraciones - Variables](#)

Primitivo/a

Un valor primitivo es un dato que no es una estructura, no tiene miembros y no es Nada.

PuréScript cuenta con 3 primitivas:

- [Número](#)
- [Texto](#)
- [Lógico](#)

Todos estos valores primitivos se sitúan al nivel más bajo de la implementación del lenguaje.

Todas las primitivas son inmutables, o sea que no se puede alterar su estado. Aquí es donde hacemos énfasis en la distinción entre un valor primitivo y una *variable* que contiene un primitivo.

Una variable de valor primitivo puede ser cargada con un nuevo valor, pero el valor contenido no puede ser manipulado en formas que se lo vería con Listas o Registros.

Cualquier ilusión de “modificación” de un valor primitivo es en realidad un nuevo valor creado a partir del original, en lugar de ser el valor original con cambios sufridos.

¿Te perdiste? Vuelve a:

- [Guía - Declaraciones – Tipos](#)
- [Guía - Funciones - Paso de Parámetros por Valor y por Referencia](#)

Estructura

Una estructura en PuréScript una variable que alberga o puede albergar un conjunto ordenado o desordenado de datos.

PuréScript cuenta con 3 tipos de estructura:

- [Lista](#)
- [Registro](#)
- [Marco](#)

Una estructura puede contener valores, variables o incluso otras estructuras.

Puedes acceder cualquier dato directamente contenido en una Lista con el operador de flecha ("->") siguiendo la sintaxis `estructura->dato`. Si el `dato` fuera también una estructura, se podrían seguir concatenando expresiones de flecha siguiendo la sintaxis:

`estructura->estructura->dato`.

¿Te perdiste? Vuelve a:

- [Guía - Declaraciones – Tipos](#)
- [Guía - Funciones - Paso de Parámetros por Valor y por Referencia](#)

Usuario

«Referente a un entorno de ejecución en Discord»

Un *Usuario* es cualquier **miembro** de servidor de Discord que usa el comando **p!tubérculo** con **Bot de Puré** para ejecutar un **Tubérculo**. Se dice que “es Usuario de ese **Tubérculo**” durante su ejecución.