PuréScript

Documentación de Lenguaje de Scripting de Tubérculos

Papita con Puré · v1.01

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.
Si no conoces algún concepto de informática, puedes revisar el capítulo de Definiciones.

Índice de Capítulo

Ilustración https://twitter.com/pu_yan8901

Introducción

PuréScript (PS) es un lenguaje de programación interpretado, funcional y dinámico para Bot de Puré (alias "Bot"). Se emplea en la creación de comandos personalizados de servidor (alias "Tubérculo") y permite computar una sucesión de órdenes para manipular y personalizar la ejecución del Tubérculo. Todo PuréScript válido debe concluir en el envío de un mensaje o valor.

PuréScript permite crear variables y estructuras complejas como listas (conjunto de valores) o glosarios (conjunto de pares de claves y valores) en tiempo de ejecución, también cuenta con funciones ciudadanas de primera clase.

Hola Mundo

El siguiente código hace que se envíe el mensaje *"¡Hola mundo!"* en el chat.

```
ENVIAR "¡Hola mundo!"
```

Gramática

La sintaxis del lenguaje se lee casi como español. Emplea elementos sintácticos de lenguajes como C, pero los aplica de una forma que se lee casi como español.

PuréScript **no distingue mayúsculas, minúsculas y tildes**. La única excepción a esto son los identificadores de variables o funciones, que deben coincidir en todo lo anterior para estar relacionados.

Sentencias

En PuréScript, las instrucciones se llaman **sentencias**, y se separan con **indicadores de sentencia**. Cada sentencia comienza con un indicador de la sentencia a procesar. Dichos indicadores se colorean de **naranja fuerte** en el código de este documento.

Los indicadores de sentencia suelen ser verbos como REGISTRAR, CARGAR, SUMAR, COMENTAR, o el previamente visto ENVIAR; y definen la estructura y comportamiento de la sentencia en cuestión.

Nótese que, si bien los indicadores de sentencia no distinguen mayúsculas y minúsculas, se recomienda como convención ponerlos todo en mayúsculas. Dicho esto, puedes usar tus propias convenciones si no te gusta, pero deberías al menos ser consistente.

Análisis Léxico e Interpretación

El texto fuente de PuréScript se lee de izquierda a derecha, y se convierte en una secuencia de símbolos llamados **Tokens**. Los espacios, "siguientes renglones" y sangrías generalmente son ignorados en este proceso, a menos que estén separando dos palabras en diferentes Tokens, o dentro de un Texto.

Por ejemplo, `CARGARTexto` no es lo mismo que `CARGAR Texto`. Sin embargo, `miembro->propiedad` es lo mismo que `miembro -> propiedad`. También, `Glosario` no se interpreta igual que `Glo sa rio` y `"Hola mundo"` no es lo mismo que `"Holamundo"`.

Las sentencias como tal son Tokens, que contienen otros Tokens, que pueden contener incluso más Tokens. Y todo script de PuréScript parte de un Token raíz llamado Programa, el cual contiene la sucesión de sentencias principales.

Comentarios

Puedes realizar comentarios sobre lo que hace tu código. La sintaxis de estos comentarios difiere de lenguajes como C. En PuréScript, la sentencia COMENTAR permite ingresar un comentario sobre lo que hace el código como un texto encerrado entre comillas dobles.

Puedes agregarlos si sientes que es necesario explicar alguna parte de tu código.

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

Declaraciones

PuréScript ofrece 2 sentencias para declarar variables (véase Variables abajo):

`CREAR`

> Declara una nueva variable del tipo especificado y no le asigna ningún valor, a menos que el tipo tenga un valor por defecto (véase Tipos abajo).

`CARGAR`

> Declara una variable y le asigna el valor especificado. Si la variable ya existía, simplemente se le asigna el valor de la expresión.

Variables

⬆ Puedes usar variables como nombres simbólicos para representar valores en tu script. Estos nombres de variable, llamados identificadores, siguen ciertas reglas:

- Siempre comienzan con una letra o guion bajo ("_").
- El resto de caracteres también pueden ser números.
- No pueden contener espacios (el espacio delimita el nombre).
- No pueden llamarse igual que una *palabra reservada*, como un indicador de sentencia, operadores o tipos de variable.
- Los identificadores son las únicas palabras en PuréScript que distinguen letras mayúsculas de minúsculas, y tildes.

Todas las declaraciones en PuréScript son de variables. Lo cual significa que no puedes declarar constantes. Los identificadores de variable se colorean de celeste en el código de este documento.

Si quieres, puedes leer más sobre variables en Wikipedia.

Tipos

⬆ PuréScript cuenta con 6 tipos de variable. 3 de los cuales representan valores básicos, y otros 3 que representan estructuras complejas. Existe el valor primitivo especial `Nada` para representar "ningún valor". Los indicadores de tipo se colorean de verde en el código de este documento.

`Número`

Tipo primitivo. Representa un número real. Por defecto es `Nada`.

`Texto`

Tipo primitivo. Representa texto, entre comillas dobles. Por defecto es `Nada`. Poner el indicador de tipo "Texto" delante de una palabra hace que se detecte esa palabra como un texto, sin necesidad de usar comillas dobles.

`Dupla`

Tipo primitivo. Representa uno de dos valores lógicos: `Verdadero` o `Falso`. Se puede alternar entre ambos con el operador `no`. Por defecto, es `Falso`.

`Lista`

Tipo de estructura. Representa un conjunto ordenado de valores/expresiones. El primer elemento es de índice 0, el segundo es de índice 1, el tercero 2, y así. Por defecto, es una lista sin elementos (ningún índice).

`Glosario`

Tipo de estructura. Representa un conjunto de pares relacionales de claves y valores. Puedes pensar en ello como un diccionario: las claves serían palabras y los valores serían definiciones.

`Marco`

Estructura compleja especial para PuréScript. Representa la estructura de un marco de mensaje (Embed) de Discord. Se utiliza solamente para enviar un marco, y sus propiedades se asignan por medio de unas funciones reservadas.

Bloques y Ámbito de Variables

Las sentencias se agrupan en **bloques**, que en sí mismos *parten de una sentencia*. Al ejecutar un Tubérculo, el primer bloque que se ejecuta es el bloque Programa, el cual puede ejecutar otros bloques que contenga.

Se expandirá en cómo crear bloques más adelante. De momento, es importante recordar que un bloque es básicamente un conjunto de sentencias.

Declaraciones y Asignaciones

Como ya vimos, la sentencia CREAR declara una variable del tipo especificado. Sin valor o con el valor por defecto del tipo indicado:

```
COMENTAR "Crea la variable 'nombre' de tipo Texto, sin valor"
CREAR Texto nombre

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

Igualmente, la sentencia CARGAR tiene la posibilidad de declarar variables, pero también las inicializa en un valor. Adicionalmente, CARGAR puede asignar diferentes valores a variables existentes:

```
COMENTAR "Declara la variable 'contador' de tipo numérico y le asigna 3"
CARGAR contador con 3

COMENTAR "Crea la lista 'frases' y le asigna 3 textos"
CREAR Lista frases
CARGAR frases con Lista "Hola", "Buenas", "Adiós"
```

Aquí notamos una nueva sintaxis, que hace uso del operador “con”.

Operador "con"

El operador `con` asigna el valor de la expresión de la derecha al identificador de la izquierda. En este tipo de expresiones, 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. Este es el proceso de asignación.

Leyendo la instrucción del ejemplo, `CARGAR contador con 3`, podemos leerlo en español como "toma una variable 'contador' y cárgala con el valor 3". Bastante directo.

Las variables siempre deben ser declaradas antes de poder asignarles un valor (lo cual se puede hacer automáticamente con CARGAR si no se usa CREAR), y deben ser asignadas un valor antes de ser utilizadas.

```
COMENTAR "Esto causa un error porque la variable 'referencia'
todavía no ha sido declarada en el punto que se la menciona"

CARGAR unValor con referencia
CARGAR referencia con 3
```

`CREAR var CARGAR var con 3` y `CARGAR var con 3` tienen el mismo comportamiento si `var` no está declarado. Sin embargo, si `var` ya se encuentra declarado en el bloque actual (Véase Ámbitos debajo) y se lo intenta volver a declarar, saltará un error.

```
COMENTAR "Se alza un error por intentar declarar 'unTexto' 2 veces"

CARGAR unTexto con "Había una vez..."
CREAR Texto unTexto
```

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

Ámbito de Variables

La declaración de variables, por medio de las sentencias CREAR o CARGAR, se realiza en un cierto *ámbito*. El ámbito de una variable define qué partes del código pueden accederla.

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

- Ámbito global: La variable es accesible por el resto del Tubérculo.
- Ámbito de bloque: La variable es accesible en el bloque actual.
- Ámbito de Función: Similar al de bloque; lo veremos después.

Una variable con ámbito de bloque será destruida cuando el mismo finalice. Declarar una variable directamente en el bloque Programa hace que la misma permanezca desde que se declara hasta que el programa termina de ejecutarse (enfoque global).

```
COMENTAR "Esta variable permanecerá viva por el resto del Programa"

CARGAR unValor con 3
```

Una variable destruída es eliminada y olvidada por completo. Lo cuál significa que puede declararse otra variable bajo el mismo nombre más adelante en el Tubérculo.

Puedes leer más sobre ámbitos de variable en Wikipedia. PuréScript usa ámbitos léxicos.

El Operador "+" con Diferentes Tipos

Textos y Otros Tipos

El operador `+` concatena expresiones cuando se encuentra con un Texto y otro tipo.

```
COMENTAR "Puntos: 100"
CARGAR unTexto con "Puntos: " + 100

COMENTAR "Tienes 100 puntos"
CARGAR unTexto con "Tienes " + 100 + " puntos"

COMENTAR "22"
CARGAR unTexto con "2" + 2

COMENTAR "4"
CARGAR unTexto con Texto 2 + 2

COMENTAR "El dicho es Verdadero"
CARGAR unTexto con "El dicho es " + Verdadero

COMENTAR "Esto es RARO"
CARGAR unTexto con "Esto es " + (Lista "R", "A", "R", "O")
```

Si se usan otros operadores binarios aritméticos con Texto (`-`, `*`, etc), se alzará un error.

Números y Otros Tipos

Si se intensa usar el operador `+` entre un Número y otro tipo que no sea un Texto, será el otro tipo el que será convertido a un Número.

```
COMENTAR "var = 3"
CARGAR var con 3 + Falso

COMENTAR "var = 1001"
CARGAR var con 1000 + Verdadero
```

Literales

Los literales representan valores, pero no son variables. Son valores fijos 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 Dupla
- Literal de Lista
- Literal de Glosario

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

Literal de Número

⬆ Los literales de Números en PuréScript incluyen enteros y decimales sin signos.

Todos los Números en PuréScript se interpretan en base-10, significando que no hay expresiones binarias ni octales ni hexadecimales. Solo decimales.

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

Puedes poner un operador unario `+` delante de un valor para convertirlo a un Número. Funciona igual con el operador unario `-`.

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

Enteros

Solo contienen dígitos.

```
0, 42, 5, 759, 02062003, 1674663241
```

Decimales

Contienen dígitos y un punto decimal.

```
0.5, 42.3, 9.999999999, 759.01, .504
```

Literal de Texto

⬆ Un literal de Texto son 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"
```

Plantillas de Texto

PuréScript también cuenta con Plantillas de Texto. Que son básicamente múltiples expresiones de Texto o convertibles a Texto separadas por coma. Puedes pensar en ello como otra forma de expresar concatenación de Textos.

```
CARGAR horas con 10
CARGAR var con "Llevamos ", horas, " horas aquí, ¿podemos irnos?"
```

En el caso de que la primer expresión no sea un literal de Texto, se debe especificar que se intenta construir una plantilla de Texto con el indicador de tipo.

```
CARGAR horas con 3
CARGAR var con Texto horas, " horas más y finalmente nos vamos..."
```

Secuencias de Escape en Textos

Las combinaciones de caracteres que constan de una barra inversa (\) seguidas de cualquier caracter se denominan "secuencias de escape". Las secuencias de escape se pueden ver como un tipo de caracter especial que señala una “acción” dentro del Texto.

Cabe mencionar que tanto el caracter “\” como el caracter subsecuente son eliminados del Texto resultante, y en cambio, se ingresa alguna de las siguientes acciones (o simplemente se ignoran los caracteres, para los casos que no se ven en la tabla):

Caracter	Acción
`\n`	“Nueva línea”. Salta al siguiente renglón.
`\”`	Coloca el caracter literal de comillas dobles en el Texto.
`\\`	Coloca el caracter literal de barra inversa en el Texto.

Ejemplos

```
"¿Ya te conté sobre la vez que conocí a \"Papita con Puré\"?"
"¡Esto es un renglón!\n¡¡Esto es otro renglón!!"
"¿Sabes? \\ Separar con estas barras puede quedar cool."
```

Literal de Dupla

⬆ Un literal de Dupla puede ser `Verdadero` o `Falso`. Se colorean de rojo en el código de este documento.

Literal de Lista

⬆ Un literal de Lista es una sucesión de 0 ó más expresiones, cada una de las cuáles representa un elemento de Lista, precedida por el indicador de tipo de Lista. Las expresiones como tal se separan con comas.

Crear una variable de Lista con un literal de Lista inicializa la variable con los valores especificados como elementos, y su `largo` con la cantidad de valores.

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

En el anterior ejemplo, se crea una Lista `juegos` con 3 elementos. En el índice 0 se contiene `"Ajedrez"`, en el índice 1 `"Damas"` y en el índice 2 `"Truco"`. Las Listas se indexan desde 0.

Para acceder a un elemento de Lista, se usa el operador de flecha `->` seguido del índice.

En el caso de `juegos`:

```
ENVIAR juegos->0 COMENTAR "Envía 'Ajedrez'"
ENVIAR juegos->1 COMENTAR "Envía 'Damas'"
ENVIAR juegos->2 COMENTAR "Envía 'Truco'"
ENVIAR juegos->largo COMENTAR "Envía '3'"
```

Puedes señalizar espacios vacíos en la Lista con comas extra. 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 COMENTAR "Envía '123NadaNada456'"
```

Literal de Glosario

⬆ Un literal de Glosario es una sucesión de 0 ó más pares de identificadores y expresiones unidos por el operador de dos puntos `:`, cada uno de los cuáles representa un miembro de Glosario, precedido por el indicador de tipo de Glosario. Los pares de identificador/expresión se separan con comas.

Crear una variable de Glosario con un literal de Glosario inicializa la variable con los pares especificados como miembros, y su `tamaño` con la cantidad de pares.

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

COMENTAR "Terraria es un juego de Acción/Aventura para PC, Consolas, Móvil.
Salió en 2011"
ENVIAR Texto juego->nombre, " es un juego de ", juego->género, " para ",
juego->plataformas->unir(", "), ". Salió en ", juego->salida
```

Como se vio en el anterior ejemplo, para acceder a los miembros del Glosario `juego` se usa el operador de flecha (“->”) seguido del identificador de un miembro. También podemos apreciar cómo el miembro `plataformas` contiene una Lista de 3 elementos. En ENVIAR nos referimos a este miembro e invocamos el método de Lista `unir` para insertar todos los elementos de la Lista unidos con comas. Es posible acceder a todos los métodos y elementos de esta Lista. Por ejemplo, si expresamos `juego->plataformas->0` veríamos el valor `"PC"`.

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.

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

COMENTAR "Esto envía '¡¡¡WOW!!!', porque la variable 'común' fue
modificada dentro de un bloque de este ámbito"
ENVIAR común

COMENTAR "Esto causa un error, 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 de este bloque se encuentra *dentro* del ámbito del bloque que lo llama, y esto aplica para múltiples bloques anidados. Esto abre las puertas a la programación estructurada.

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. Es bastante simple, e incluso más si lees el código en sí.

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:

```
CREAR Texto útil
BLOQUE
    BLOQUE
        CARGAR café con "colombiano"
        BLOQUE
            CARGAR algo con "No sé en dónde me metí..."
            ENVIAR "Me gusta mucho el café ", café, "."
        FIN
    FIN
    ENVIAR "¿", algo, "? Eso ya no existe."
    CARGAR inútil con "Esto es inútil."
    CARGAR útil con "Esto es útil."
FIN
ENVIAR "¿Qué es ", inútil, "? Yo solo conozco a ", útil, "."
CREAR Número inútil
CARGAR inútil con 23
ENVIAR Texto inútil, " es el vigésimo-tercer número."
```

En este ejemplo, se pueden hacer varias observaciones. Mira con detalle:

- 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 hijo, ya que este bloque hijo como tal está 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` en forma de Texto.

Múltiples Variables con el Mismo Identificador

Así como este código es válido:

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

También podemos hacer algo muy similar con variables de mismo nombre existiendo al mismo tiempo en diferentes ámbitos.

Veamos el siguiente ejemplo. 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.

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

Esto nos permite declarar varias variables bajo el mismo identificador simultáneamente, lo cuál nos será bastante 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 tiene prioridad sobre la de un ámbito más general como Programa.

Y bueno, esto es genial, pero la sentencia BLOQUE por sí sola no es de tanta utilidad, y es la sentencia de bloque más básica que hay. Esto fue a modo de terminar de explicar cómo funcionan los bloques, ¡así que vamos ahora con lo bueno! Existen otras sentencias de bloque que modifican el flujo del bloque, de una forma u otra. Estas se conocen como **estructuras de control** (puedes buscar luego más información en Wikipedia).

Estructuras de Control

Las sentencias de estructura de control se colorean de **naranja oscuro** en el código de este documento. A continuación, veremos las siguientes estructuras de control:

- Condicionales
  - SI
  - SI...SINO
- Iterativas
  - MIENTRAS
  - HACER...MIENTRAS
  - PARA

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.

La sentencia SI

⬆ La sentencia SI toma una expresión lógica (profundizaremos más abajo) como condición, y ejecuta el bloque que le sigue si evalúa a `Verdadero`. Si no, se lo ignora.

```
SI Falso
    COMENTAR "Las sentencias escritas aquí dentro se ignorarán"
FIN
SI Verdadero
    COMENTAR "Las sentencias escritas aquí dentro se ejecutarán"
FIN
```

La sentencia SINO y la estructura SI...SINO

⬆ La sentencia de bloque SINO define un bloque que se ejecutará si el bloque SI que lo precede no se ejecuta. Esto crea una estructura SI...SINO, que en español hablado se lee algo como "Si esto es verdad, haz esto. Si no, haz esto otro".

Expandiendo del ejemplo anterior, quedaría así:

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

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

También, puedes especificar otra condición inmediatamente después del SINO:

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

Expresiones Lógicas y Operadores Lógicos

Una expresión lógica toma un conjunto de expresiones y devuelve una Dupla (eso es, `Verdadero` o `Falso`) dependiendo de la evaluación de las expresiones contenidas. Las expresiones contenidas se operan por medio de operadores lógicos. Puedes pensar en ello como "cálculos matemáticos pero para valores lógicos". Muy cool.

Tomemos del ejemplo la expresión lógica `2 es 4`.

Operador "es"

El operador `es` toma las dos expresiones a sus costados, en este caso `2` y `4`, y evalúa `Verdadero` si ambas expresiones son iguales (valen lo mismo). Como vimos en el ejemplo anterior, la expresión `2 es 4` evalúa a `Falso` y hace que el bloque subsecuente se ignore, porque obviamente 2 no vale lo mismo que 4.

Operador "no"

Este operador toma la expresión que le sigue y la "invierte". O sea que si evaluaba `verdadero`, ahora va a evaluar `falso`. Es un operador unario, y funciona similar a como lo haría el operador "-" con Números.

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

SI no Verdadero
    COMENTAR "'no Verdadero' es 'Falso', así que esto se ignora"
FIN
```

Operador "no es"

Este operador es como una combinación entre los operadores "no" y "es". Hace lo mismo que "es" pero invierte el resultado de la evaluación.

```
SI 2 no es 4
    COMENTAR "Esto se ejecuta, porque en efecto: 2 no es 4"
FIN
```

Hasta ahora vimos los operadores de igualdad y el operador no, pero no hemos visto ningún operador de comparación. Veámoslos.

Operador "excede"

Evalúa si la expresión de la izquierda es de un valor mayor al de la derecha. Exactamente lo mismo que expresar en matemáticas *"$a > b$"*, donde a y b son las expresiones de la izquierda y la derecha respectivamente.

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

Operador "precede"

Evalúa si la expresión de la izquierda es de un valor menor al de la derecha. Exactamente lo mismo que expresar en matemáticas *"$a < b$"*, donde a y b son las expresiones de la izquierda y la derecha respectivamente.

```
SI 2 precede 4
    COMENTAR "Esto sí se ejecuta, 2 es menor que 4"
FIN
```

Así como existe el operador "no es" para combinar "es" con "no", existen los operadores "no precede" y "no excede" para referirse a *"$a \geq b$"* y *"$a \leq b$"* respectivamente.

Precedencia de operadores

Al igual que en matemáticas, los operadores listados arriba tienen una cierta precedencia, que determina el orden de operaciones. Así como la multiplicación se hace antes que la suma, la inversión con "no" se evalúa antes que la igualdad "es". Las operaciones que van antes tienen **mayor precedencia**. Estas son las precedencias lógicas:

"no" > ("excede"/"precede") > ("es"/"parece") > "y" > "o"

Volveremos al tema de operadores y expresiones más adelante, así que estate tranquilo.

Estructuras Iterativas

⬆ Ahora que ya hicimos ese pequeño hincapié sobre los operadores lógicos, sigamos con las estructuras de control. Ahora veremos las estructuras iterativas, que siguen la misma lógica que las estructuras condicionales, pero se repiten (*iteran*) en base a la condición.

La sentencia MIENTRAS

⬆ Esta sentencia, como su nombre lo indica, ejecuta su bloque **mientras** la condición evalúe a `Verdadero`.

Veamos la sintaxis:

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

Esto, además de ser un bloque condicional, también se denomina "bloque iterativo". Ejecuta su bloque contenido múltiples veces, hasta que su condición deja de cumplirse.

La condición debería dejar de cumplirse eventualmente al modificar los componentes de la condición dentro de la ejecución del bloque. Si no se modifica nunca hasta evaluar a `Falso`, hablamos de un bucle infinito, lo cual alzará un error luego de un número de ejecuciones.

Límites a considerar

Una ejecución de Tubérculo está limitada a 600 sentencias procesadas. Eso generalmente es más que suficiente, pero si se supera este límite, se alzará un error para proteger a Bot.

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.

Las sentencias HACER y YSEGUIR, y la estructura HACER...MIENTRAS

⬆ El funcionamiento de la estructura es el mismo que el de MIENTRAS, con la diferencia de que el bloque siempre se va a ejecutar *al menos una vez* (incluso si la condición nunca evalúa a `Verdadero`). Esto se debe a que se evalúa la condición *luego* de ejecutar el bloque.

También, en lugar de señalizar el fin del bloque con FIN, se usa una sentencia especial llamada YSEGUIR, seguida inmediatamente de el indicador MIENTRAS y una expresión.

La sintaxis tendrá más sentido si la vemos en código. Veamos el siguiente ejemplo:

```
CARGAR ejecuciones con 100
HACER
    COMENTAR "Esto se va a ejecutar 100 veces"
    COMENTAR "En cada ejecución, se restará 1 a 'ejecuciones'"
    RESTAR ejecuciones
YSEGUIR MIENTRAS ejecuciones excede 0

COMENTAR "Estamos seguros de que 'ejecuciones' vale 0 en este punto"

HACER
    COMENTAR "Esto se va a ejecutar 1 vez"
    COMENTAR "Incluso si 'ejecuciones' no excede 0"
    RESTAR ejecuciones
YSEGUIR MIENTRAS ejecuciones excede 0

COMENTAR "En este punto, 'ejecuciones' vale -1"
```

El código se explica bastante solo, pero básicamente estamos primero ejecutando el contenido de los bloques iterativos y **después** comprobando la condición. Un pequeño cambio en el orden que puede llegar a afectar bastante el programa.

MIENTRAS es generalmente más utilizado que HACER...MIENTRAS. Es preferible, de hecho. Sin embargo, debes tener la capacidad de considerar ambas estructuras cuando quieras hacer una tarea repetidas veces. Ambas son importantes.

La sentencia PARA

⬆ Antes que explicar directamente cómo funciona PARA, será mejor revisar un ejemplo:

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

Aquí estamos:

1. Declarando una variable `ejecuciones` antes de comenzar el bloque.
2. Comprobando si `ejecuciones excede 0` antes de cada iteración del bloque.
3. Restando a `ejecuciones` al final de cada iteración del bloque.

La sentencia PARA hace todo esto de forma más compacta. Inicializa un contador en el ámbito del bloque que ejecuta, ejecuta dicho bloque mientras su condición sea verdadera y actualiza el valor del contador en cada iteración. Esto nos deja con la siguiente sintaxis:

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

Como podemos ver, se lee bastante natural y posee un funcionamiento idéntico al primer ejemplo con MIENTRAS. Y podrías decir que es exactamente lo mismo que expresar "`CARGAR ejecución con 100 MIENTRAS ejecución excede 0 RESTAR ejecución`", pero hacer eso con un MIENTRAS haría que la variable `ejecución` siga existiendo fuera del bloque iterativo. Eso se puede solucionar usando un BLOQUE por fuera, pero la estructura PARA se lee más bonita.

Otras Sentencias de Control

La 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 la ejecución del Programa de forma temprana, o dentro de un bloque de Función para terminar la Función (veremos funciones más abajo).

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

La estructura PARA CADA

En el caso de querer iterar sobre los elementos de una Lista, 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"

COMENTAR "Envía el mensaje con el nombre de todos los gnomos"
PARA CADA gnomo en 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` de la iteración actual.

Esta estructura puede verse como un parafraseo de la estructura PARA, pero no te confundas. PARA CADA es de mucha ayuda, ya que te ahorra los pasos de usar una variable contador para acceder a cada elemento de la Lista con el uso del operador de flecha. En esta estructura, simplemente especificas la Lista (`gnomos`) y cómo quieres referirte al elemento de cada iteración (`gnomo`). Si quieres iterar sobre una Lista y no te importa el índice, esta es generalmente la mejor opción.

Funciones

Las funciones son esenciales en cualquier lenguaje de programación de este tipo. Una función en PuréScript es básicamente un conjunto de sentencias (bloque) que realiza una determinada tarea o calcula un cierto valor cuando se la invoca. Puede devolver un valor a la expresión que la invoca, modificar el estado del programa o ambas.

Llamado y Ámbito de Función

Una función no se ejecuta inmediatamente al declararse, sino que se la puede referenciar luego para ejecutarla, casi como si fuera una variable. "Llamar" a una función la ejecutará con los argumentos proveídos, y puede devolver un valor a la expresión que la llama.

Las funciones, al igual que las variables, respetan un cierto ámbito en el cual pueden llamarse.

Declaración de Funciones

Una declaración o registro de función se realiza mediante la sentencia REGISTRAR, seguida por:

1. El indicador de tipo de Función
2. El identificador que se le desea asignar
3. Una secuencia de **argumentos** entre paréntesis, separados por comas.
4. Una secuencia BLOQUE definiendo el comportamiento de la función, terminado en la secuencia FIN.

Los indicadores de Función se colorean de amarillo claro en el código de este documento. También pueden identificarse por estar seguidos de paréntesis.

Veamos un ejemplo sencillo. La función `esPar` toma un argumento (también conocido como parámetro), llamado `núm`, y devuelve `Verdadero` si el Número `núm` ingresado es par. La sentencia DEVOLVER indica un valor a devolver e interrumpe la ejecución del resto de la función:

```
REGISTRAR Función esPar(núm) con BLOQUE
    SI núm % 2 es 0
        DEVOLVER Verdadero
    SINO
        DEVOLVER Falso
    FIN

    COMENTAR "Esto nunca se ejecuta debido a los DEVOLVER de arriba"
FIN

COMENTAR "Envía 'Verdadero'"
ENVIAR esPar(42)

COMENTAR "Envía 'Falso'"
ENVIAR esPar(11)
```

"Devolver un valor" puede interpretarse como "colocar" el resultado de una función en donde se la llama. `ENVIAR esPar 42` termina siendo `ENVIAR Verdadero`, por ejemplo.

Cuando pasas un valor literal o una variable primitiva a una función, se realiza una "copia" del valor en el ámbito de la función. Modificar esta copia en el cuerpo de la función no afectará a la variable original, y la copia en sí se vuelve parte privada de la función:

```
REGISTRAR Función fn(var) con BLOQUE
    COMENTAR "La variable 'var' aquí es una copia de lo ingresado"
    CARGAR var con 30
FIN

CARGAR var con 15

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

COMENTAR "Envía 15, porque la 'var' original nunca fue modificada"
ENVIAR var
```

Por hacer una analogía: imagina que `var` es un papel con instrucciones y la función `fn` es tu amigo Bob que está dispuesto a ayudarte con eso. *Llamas* a Bob, sacas una fotocopia del papel con instrucciones y le entregas la fotocopia. Esperas a que Bob vuelva de hacer lo que le pediste y sigues con lo tuyo. La fotocopia de Bob podría estar llena de anotaciones y tachones como resultado de realizar las tareas, pero tu copia original no fue modificada.

Esto se conoce como "paso de parámetros por valor". Caso en el cual los argumentos de una función se pasan *por valor* a la misma.

Paso de parámetros por valor y referencia

Dijimos que el paso de parámetros por valor aplicaba a literales y datos primitivos, pero no hemos hablado de lo que ocurre al pasar una estructura como una Lista o un Glosario a una función. Observemos:

```
REGISTRAR Función agregarGorila(li) con BLOQUE
    COMENTAR "La variable 'li' es realmente el argumento pasado"
    COMENTAR "EXTENDER agrega un nuevo elemento a la Lista"
    EXTENDER li con "Gorila"
FIN

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

COMENTAR "Se envía una referencia de la Lista 'primates'..."
EJECUTAR agregarGorila(primates)

COMENTAR "Envía 'Mono, Orangután, Lémur, Gorila'"
ENVIAR primates->unir(", ")
```

En este caso, volviendo a la analogía: le diste a Bob tu lista del supermercado y te la devolvió con tachones (y obviamente te trajo los productos que encargaste).

Pasar una variable por referencia hace que, si la variable se modifica en la función, el cambio se vea reflejado fuera de la misma.

Literales y datos primitivos: se pasan por valor. **Variables de estructuras complejas**: se pasan por referencia. Es importante que todo esto se entienda, ya que las funciones son fundamentales. Asegúrate de comprender esta sección antes de continuar.

Expresiones de Función

La declaración de funciones de arriba se hacía por medio de la sentencia REGISTRAR, pero las funciones en PuréScript pueden funcionar como cualquier otra variable. Las funciones también pueden crearse con expresiones de función.

Estas funciones pueden ser **anónimas**, o sea que no necesitan tener un nombre. Volviendo al ejemplo de `esPar`, podríamos haber hecho que la función se definiera así:

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

    COMENTAR "Esto nunca se ejecuta debido a los DEVOLVER de arriba"
FIN

COMENTAR "Envía 'Verdadero'"
ENVIAR esPar(6)
```

Nótese cómo luego del indicador de tipo `Función` simplemente se declara el argumento `núm` **entre paréntesis**. Esto señala que queremos crear una Función sin nombre. La función como tal luego se carga a la variable `esPar` para ser llamada.

Sin embargo, puedes darle un nombre a una expresión de Función si quieres. Esto permite, por ejemplo, la posibilidad de que la Función se llame a sí misma de otra forma.

```
CARGAR fac con Función factorial(núm) BLOQUE
    SI núm excede 1
        DEVOLVER núm * factorial(núm - 1)
    SINO
        DEVOLVER núm
    FIN
FIN
COMENTAR "Envía 6! = 6 * 5 * 4 * 3 * 2 * 1 = 720"
ENVIAR fac(6)
```

Por cierto, estas funciones que se ejecutan a sí mismas se conocen como “funciones recursivas”. Funcionan similar a un bloque iterativo.

Volviendo con las expresiones de función: pueden ser convenientes cuando se quiere pasar una función como parámetro de otra.

En el siguiente ejemplo, `cambiarLista` es una Función que debe tomar una Lista `li` y una función `fn`, y llama `fn` para cada elemento de `li`. Recordemos el paso por referencia:

```
REGISTRAR Función cambiarLista(li, fn) con BLOQUE
    CREAR Lista resultados
    PARA CADA elemento en li
        EXTENDER resultados con fn(elemento)
    FIN
    CARGAR li con resultados
FIN
```

Con esto en mente, continuemos y creemos una variable `duplicar`, que representa una función que devuelve el doble de lo que ingreses, y pasémosla a `cambiarLista` en conjunto con una Lista de números (`unaLista`). Esto se traducirá a que todos los elementos de `unaLista` sean duplicados:

```
REGISTRAR Función cambiarLista(li, fn) con BLOQUE
    CREAR Lista resultados
    PARA CADA elemento en li
        EXTENDER resultados con fn(elemento)
    FIN
    CARGAR li con resultados
FIN

CARGAR duplicar con Función (val) BLOQUE
    DEVOLVER val * 2
FIN

CARGAR unaLista con Lista 6, 2, 4, 7, 10
EJECUTAR cambiarLista(unaLista, duplicar)
COMENTAR "Envía '12, 4, 8, 14, 20'"
ENVIAR unaLista->unir(", ")
```

Ahora que sabemos que las Funciones pueden ser simplemente una variable más, consideremos el siguiente código:

```
REGISTRAR Entrada acción con "gritar"

CREAR Función hacerAlgo

SI acción es "gritar"
    CARGAR hacerAlgo con Función () BLOQUE
        ENVIAR "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    FIN
SINO SI acción es "correr"
    CARGAR hacerAlgo con Función () BLOQUE
        ENVIAR "1... 2... 3... hasta cuándo tengo que correr..."
    FIN
SINO
    CARGAR hacerAlgo con Función () BLOQUE
        ENVIAR "¿...no se te ofrece nada?"
    FIN
FIN

COMENTAR "Realiza una acción diferente dependiendo de la entrada"
EJECUTAR hacerAlgo
```

Aquí, se ejecuta una función `hacerAlgo` diferente dependiendo de la acción ingresada por el Usuario (explicado luego). Hacer esto de esta forma no sería posible con REGISTRAR.

Métodos

Un método es una función que es propiedad de una variable. Como ya vimos repetidas veces antes, solo que sin explicarlo del todo, las Listas tienen el método `unir`. Los métodos, al igual que los elementos de Lista o miembros de Glosario, se acceden mediante el operador de flecha. `unir` básicamente devuelve un Texto con todos los elementos de la Lista, unidos con el Texto pasado de argumento entre cada uno. En este caso, `", "`.

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

COMENTAR "Envía 'Mono, Orangután, Gorila, Lémur'"
ENVIAR primates->unir(", ")
```

Llamado de Funciones

Profundicemos en el llamado de funciones. Como se mencionó previamente, *declarar* funciones no las *ejecuta*. Definirla simplemente guarda un identificador y le dice a PuréScript qué hacer cuando se lo llama.

El **llamar** la función es lo que ejecuta las sentencias descritas junto a los parámetros indicados. Por ejemplo, si defines la Función `esPar`, podrías llamarla así:

```
esPar(3)
```

La anterior instrucción llama a la función con un argumento `3`. La función ejecuta las sentencias que contiene y retorna `Falso`. Cada ejecución tiene su propio contexto.

Los argumentos no se limitan a solo Números y Textos, sino que pueden ser Listas, Glosarios o incluso Marcos. La función `marcoAsignarTítulo` es un ejemplo de una función que toma un Marco como argumento (por referencia), y le asigna un título (por valor).

```
EJECUTAR marcoAgregarTítulo(unMarco, "Un Título de Ejemplo")
```

Retomando un ejemplo anterior, una Función puede llamarse a sí misma. Esto introduce el concepto de *recursividad*: se dice que una Función es recursiva cuando, en su ejecución, se llama a sí misma. Aquí, la Función `factorial` calcula factoriales de forma recursiva:

```
REGISTRAR Función factorial(núm) con BLOQUE
    SI núm excede 1
        DEVOLVER núm * factorial(núm - 1)
    SINO
        DEVOLVER núm
    FIN
FIN
```

Esto te permitiría calcular los factoriales de cualquier número. Nótese que la función se ejecuta más veces mientras mayor sea el número:

```
ENVIAR factorial(1) COMENTAR "Se llama 1 vez,   envía 1         = 1"
ENVIAR factorial(2) COMENTAR "Se llama 2 veces, envía 1*2       = 2"
ENVIAR factorial(3) COMENTAR "Se llama 3 veces, envía 1*2*3     = 6"
ENVIAR factorial(4) COMENTAR "Se llama 4 veces, envía 1*2*3*4   = 24"
ENVIAR factorial(5) COMENTAR "Se llama 5 veces, envía 1*2*3*4*5 = 120"
```

Cada llamado de Función se agrega a una “pila de ejecución”. Cuando una Función se llama a otra, pospone el resto de sentencias en su ejecución hasta que termine de ejecutarse la Función que llamó. Este concepto también aplica para cuando una función se llama a sí misma.

O sea que cuando expresamos `factorial(5)`, en algún punto de la ejecución vamos a tener 4 funciones esperando un que otra función termine de ejecutarse.

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

La recursividad puede ser idéntica a un bloque iterativo en algunos casos, pero hay ocasiones en las que conviene usar una función recursiva sobre lo otro:

```
REGISTRAR Función recorrer(nodo) con BLOQUE
    SI esNada(nodo)
        TERMINAR
    FIN

    COMENTAR "Aquí harías algo con el nodo..."

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

Este ejemplo es más fácil de expresar con recursión que con solo estructuras iterativas.

Ámbito de Función

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

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

REGISTRAR Función sumarExterno() con BLOQUE
    DEVOLVER a + b
FIN
COMENTAR "Envía 1 + 2 = 3"
ENVIAR sumarExterno()

REGISTRAR Función sumarMixto() con BLOQUE
    CREAR Número a CREAR Número b
    CARGAR a con 10
    CARGAR b con 20
    DEVOLVER a + b + c
FIN
COMENTAR "Envía 10 + 20 + 30 = 60"
ENVIAR sumarMixto()
```

Funciones Anidadas

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

```
REGISTRAR Función vivir() con BLOQUE
    REGISTRAR Función comer() con BLOQUE
        ENVIAR "ñOM ÑOM ÑOM ÑOM ÑOM ÑOM ÑOM ÑOM ÑOM ÑO"
    FIN
    REGISTRAR Función dormir() con BLOQUE
        ENVIAR "ZzZzzZzZzzZzZzZzZzZzZzZzZzZzZzZzZzZz............."
    FIN
    COMENTAR "Aquí se puede llamar 'vivir', 'comer' y 'dormir'"
    EJECUTAR comer
    EJECUTAR dormir
FIN
COMENTAR "Aquí solo se puede llamar 'vivir', lo cuál envía 2 mensajes"
EJECUTAR vivir
```

Como vemos, el cuerpo de la función `vivir` hace que se declaren las funciones `comer` y `dormir` al ejecutarla. Esto significa que estas 2 funciones no están declaradas ni son llamables a menos que se ejecute `vivir`.

Funciones como retorno

Puedes asignar una función en lugar de su resultado si no colocas los paréntesis:

```
CARGAR fac con factorial
```

También puedes usar DEVOLVER con el identificador de una Función para devolverla:

```
DEVOLVER factorial
```

Veamos ambos casos en el siguiente ejemplo:

```
REGISTRAR Función unaFunción() con BLOQUE
    REGISTRAR Función otraFunción() con BLOQUE
        ENVIAR "Esternocleidomastoideo"
    FIN
    DEVOLVER otraFunción
FIN
CARGAR fn con unaFunción()
COMENTAR "Se envía 'Esternocleidomastoideo'"
EJECUTAR fn
```

Pequeña intermisión...
¿Te gusta cómo se ve el código PuréScript de este documento? Lamentablemente, el código en Discord no se va a ver igual de bonito, pero puedes probar este formato:

p!t -cs unTubérculo
```arm
*tu código aquí*
```

Múltiples Funciones Anidadas

Retomando un 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.

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

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

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"
REGISTRAR Función fn(var) con BLOQUE
    ENVIAR var
FIN

COMENTAR "Esto envía 'chocolate'"
EJECUTAR fn("chocolate")

COMENTAR "Esto envía 'caramelo'"
ENVIAR var
```

¡Con esto debería quedar claro! Las funciones internas pueden acceder al ámbito de las externas, pero las externas no pueden acceder al ámbito de las internas. Usa funciones siempre que lo veas conveniente. Permite que tu código sea más fácil de leer, más fácil de modificar y mantener, y evita reutilizar tanto código (créeme, si piensas programar con el límite de 2000 caracteres de Discord, cada caracter cuenta).

Argumentos de Función

Ya dijimos brevemente lo que son los argumentos de Función. Sin embargo, echemos un vistazo a los parámetros por defecto.

Parámetros por Defecto

PuréScript permite ingresar la cantidad de argumentos que quieras en un llamado de Función. Cuando ingresas menos parámetros que los que especifica la declaración de función, los parámetros no ingresados serán `Nada` durante esa ejecución.

Los parámetros por defecto te permiten asignar un valor por defecto a un argumento de Función que no se ingrese. Esto te aliviana la tarea de comprobar manualmente si un valor `esNada`.

Primero veamos un ejemplo de una Función sin parámetros por defecto.

```
REGISTRAR Función producto(a, b) con BLOQUE
    SI esNada(b) CARGAR b con 0 FIN
    DEVOLVER a * b
FIN
```

En la anterior Función, si no se pasa `a`, se alza un error; pero si no se pasa `b`, se le asigna `0`.

Ahora repliquemos este comportamiento con parámetros por defecto:

```
REGISTRAR Función producto(a, b: 0) con BLOQUE
    DEVOLVER a * b
FIN
```

Como vemos, reduce el código y es más fácil de leer.

Funciones predefinidas

Las siguientes son funciones por defecto que son siempre accesibles en un Tubérculo:

`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`.

`esDupla(expresión)`

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

`esLista(expresión)`

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

`esGlosario(expresión)`

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

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.

Expresiones y Operadores

En esta parte se verán a detalle estos "operadores" y "expresiones" de los que tanto hemos estado hablando. Probablemente ya te haces una idea de lo que estamos hablando cuando se tiran estas 2 palabras por aquí y allá, pero ahora vamos a definirlas bien.

Entonces... ¿qué es una expresión? De primeras, una expresión es cualquier pedacito de código que se resuelve en un valor. Las expresiones en PuréScript no tienen efectos secundarios por sí solas, sino que simplemente *evalúan* hacia un resultado.

Por ejemplo, en el código `CARGAR a con b` tenemos la *expresión* `a con b`. Esta expresión usa el *operador* `con`, en conjunto con la sentencia CARGAR, asigna el valor de `b` a la variable `a`. La expresión en sí evalúa hacia `b`.

Sin embargo, tomemos la expresión `a + b`, sin ninguna sentencia. Esta expresión usa el operador `+` para sumar `a` con `b`, pero el resultado se descartará inmediatamente a menos que la expresión forme parte de una sentencia que realice un cambio en el programa con dicha expresión.

```
REGISTRAR Entrada núm con 2 + 3
```

Nótese que `a` y `b` también son expresiones. Al unir ambas con un operador como `con` o `+`, se forma una *expresión compleja*. Las expresiones sin operadores ni "sub-expresiones" se conocen como *expresiones simples*.

También, es importante mencionar que el mismo operador puede representar diferentes operaciones dependiendo del contexto. Tómese por ejemplo el operador de asignación `+` con Números contra el operador de concatenación `+` con Textos.

Los operadores se colorearán de gris claro para el código de este documento, lo cual también es el color por defecto del código del documento.

A continuación, recorreremos los siguientes operadores:

- Operador de asignación
- Operadores de comparación
- Operadores lógicos
- Operadores aritméticos
- Operador de concatenación
- Operadores de agrupación

Estos operadores unen operandos, que pueden ser expresiones de mayor precedencia o expresiones simples. Puedes encontrar un listado completo y detallado de todos los operadores y expresiones en la Referencia.

Importancia de la Precedencia de Operadores

La precedencia de operadores define qué operaciones tienen prioridad sobre otras (literalmente "preceden" a otras) a la hora de evaluar una expresión. Por ejemplo:

```
CARGAR a con 1 + 2 * 3
CARGAR b con 2 * 3 + 1
```

Incluso si `+` y `*` vienen en diferente orden, ambas variables valdrían `7`. Esto ocurre porque la multiplicación (`*`) y división (`/`) tienen mayor precedencia que la suma (`+`) y la resta (`-`), así que las expresiones unidas por `*` siempre se evaluarán primero. Puedes evitar la precedencia de operadores usando paréntesis, lo cuál agrupa expresiones. Tal y como en matemáticas. El listado de niveles de precedencia también se encuentra en la Referencia.

Ya mencionamos previamente la existencia de operadores unarios en PuréScript. También existen operadores ternarios en otros lenguajes, pero en PuréScript eso se sacrifica por simplicidad. Los operadores unarios y binarios forman *expresiones unarias y binarias*.

Nota: los "< >" en la siguiente parte encapsulan términos o conceptos a modo de ejemplo.

Un operador binario requiere 2 operandos. Uno a la izquierda y otro a la derecha:

```
<operando1> <operador> <operando2>
```

En esta forma de operación, hablamos de un "operador binario infijo", porque está entre medio de los 2 operandos. Todos los operadores binarios en PuréScript son infijos.

Un operador unario requiere 1 operando, ya sea a su izquierda o su derecha:

```
<operador> <operando>
<operando> <operador>
```

En el caso de PuréScript, solo hay operadores unarios de los que van antes del operando. Este tipo de operador unario se llama "operador unario prefijo", a diferencia del "operador unario sufijo".

Operador de Asignación

⬆ Un operador de asignación indica que el operando de la izquierda recibe un valor basado en el operando de la derecha. Puedes leer esto en la Referencia.

PuréScript cuenta con un solo operador de asignación: el operador `con`. Este operador indica que, de una forma u otra, se debe asignar el valor del operando derecho al operando izquierdo. La forma en la que se asigna el valor depende de la sentencia actual. La forma más básica es una asignación directa, en la que el operando de la izquierda pasa a valer lo mismo que el operando de la derecha. Expresemos este caso como `a = b`.

Veamos la siguiente tabla:

Sentencia	Fórmula	Significado
`CARGAR a con b`	`a = b`	Asigna el valor del operando derecho al izquierdo.
`SUMAR a con b`	`a = a + b`	Suma el valor del operando derecho al izquierdo.
`RESTAR a con b`	`a = a - b`	Resta el valor del operando derecho al izquierdo.
`MULTIPLICAR a con b`	`a = a * b`	Multiplica al operando izquierdo por el valor del operando derecho.
`DIVIDIR a con b`	`a = a / b`	Divide el operando izquierdo por el valor del operando derecho.

Nótese que en este caso se ve *modificada* la variable `a`. La variable `b` simplemente es *evaluada* para ello.

Asignación de Propiedades

Si la expresión de la izquierda se evalúa como una Lista o un Glosario, entonces se puede asignar a una de sus propiedades:

```
CREAR Lista unaLista CREAR Glosario unGlosario
CARGAR unaLista->0 con 42
CARGAR unGlosario->unMiembro con 96
ENVIAR unaLista->0           COMENTAR "Envía '42'"
ENVIAR unGlosario->unMiembro COMENTAR "Envía '96'"
```

Nótese que si intentas hacer lo mismo con una variable primitiva, se alzará un error:

```
CREAR Número unNúmero
CARGAR unNúmero->wasd con 42
```

Sustitución de Valores

Cuando se usa "con" en el contexto de CARGAR, puede funcionar de dos maneras: como una *declaración* o como una *asignación*.

```
CARGAR var con 42
ENVIAR Texto var COMENTAR "Envía '42'"

CARGAR var con Lista "a", "b", "c"
ENVIAR Texto var    COMENTAR "Envía 'abc'"
ENVIAR Texto var->1 COMENTAR "Envía 'b'"
```

Operadores de Comparación

⬆️ Un operador de comparación compara sus operandos y devuelve un valor lógico (una Dupla) basado en si la comparación es verdadera o no.

Los operandos pueden ser de cualquier tipo. 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 (por ejemplo: "a" < "b").

La comparación de igualdad se hace teniendo en cuenta el tipo y el valor de los operandos, o sea que el Texto "24" no es lo mismo que el Número 24. La única excepción a esto son los operadores `parece` y `no parece`, aunque se recomienda no hacer uso excesivo de estos.

Cuando no se evalúa `Verdadero`, se puede asumir que se evalúa `Falso`.

Nótese que aquí hay operadores que consisten de más de 1 palabra ("no es"/"no parece").

A modo de relacionar mejor este concepto con usos prácticos, tomemos estas 2 variables:

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

Veamos la siguiente tabla, puedes ver algunos ejemplos si quieres:

Operador	Evaluación	Ejemplos evaluando...	
		Verdadero	Falso

`es`	`Verdadero` si los operandos son iguales, tanto en tipo como en valor.	`a es 2`
		`2 es 2`
		`"hola" es "hola"`
			`a es 4`
			`2 es "2"`

`no es`	`Verdadero` si los operandos no son iguales, tanto en tipo como en valor.	`a no es "2"`
		`2 no es "2"`
		`"x" no es "y"`
			`a no es 2`
			`a no es a`
			`"m" no es "m"`

`parece`	`Verdadero` si los operandos son iguales en valor. Incluye conversión de tipo.	`a parece 2`
		`a parece "2"`
		`2 parece "2"`
			`a parece b`
			`"x" parece "y"`

`no parece`	`Verdadero` si los operandos no son iguales en valor. Incluye conversión de tipo.	`a no parece 4`	`a no parece "2"` `2 no parece "2"`
`excede`	`Verdadero` si el operando de la izquierda vale más que el de la derecha.	`b excede a` `"12" excede 2`	`a excede b` `2 excede 2`
`no excede`	`Verdadero` si el operando de la izquierda vale menos o igual que el de la derecha.	`a no excede b` `2 no excede 2`	`"4" no excede 2`

`precede`	`Verdadero` si el operando de la izquierda vale menos que el de la derecha.	`a precede b`
		`"a" precede "b"`
			`b precede a`
			`3 precede 3`
			`"z" precede "a"`

`no precede`	`Verdadero` si el operando de la izquierda vale más o igual que el de la derecha.	`b no precede a` `a no precede 2`	`a no precede b`

Esta tabla explica brevemente el funcionamiento de los operadores de comparación.
Puedes aprender más en la Referencia.

Operadores Lógicos

⬆ Los operadores lógicos suelen usarse con valores de tipo Dupla. En estos casos, la operación también devuelve una Dupla. Sin embargo, los operadores listados aquí realmente devuelven el valor de uno de sus operandos, así que pueden devolver valores que no sean Duplas.

Por empezar, veamos la siguiente tabla:

Operador	Uso	Descripción
`y`	`a y b`	Devuelve el operando `a` si este evalúa a `Falso`. De lo contrario, devuelve `b`. Si se usa con Duplas, hace que se devuelva `Verdadero` si ambos operandos son `Verdadero`. De lo contrario, devuelve `Falso`.
`o`	`a o b`	Devuelve el operando `a` si este evalúa a `Verdadero`. De lo contrario, devuelve `b`. Si se usa con Duplas, hace que se devuelva `Verdadero` si cualquier operando es `Verdadero`. De lo contrario, devuelve `Falso`.
`no`	`no a`	Devuelve `Falso` si su operando puede evaluar a `Verdadero`. De lo contrario, devuelve `Verdadero`.

Expresiones que pueden evaluar a `Falso` en este contexto serían `0`, `""` o `Nada`.

Ahora veamos unos ejemplos para que tenga más sentido:

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

Puedes aprender más sobre operadores lógicos en la Referencia. De todas formas, cubriremos algunos otros conceptos importantes con ellos a continuación.

Evaluación de Cortocircuito

Como las expresiones lógicas 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` no se evalúa en estas 2 expresiones, así que si `algo` fuera un llamado de Función por ejemplo, no se ejecutaría.

Veamos unos ejemplos más:

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

Y sí, como te estarás imaginando si prestaste atención, los operadores lógicos y de comparación se suelen usar juntos para bloques condicionales, como por ejemplo:

```
REGISTRAR Entrada valor con 3

SI valor excede 1000000
    ENVIAR "¡Wow! ¡Ese es un gran número!"
SINO SI valor no precede 0 y valor no excede 9
    ENVIAR "Eso es solo un dígito..."
SINO
    ENVIAR "Ese es un número regular, ¿supongo?"
FIN
```

Operadores Aritméticos

⬆ Los operadores aritméticos toman expresiones numéricas (ya sean Números o variables) como operandos y devuelven un solo valor numérico. Las operaciones aritméticas más comunes son la suma (`+`), la resta (`-`), la multiplicación (`*`) y la división (`/`). Lee más aquí.

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

En adición a estas operaciones, PuréScript ofrece otras 3 de utilidad:

Operador	Descripción	Ejemplos
`%`	Operador binario. Devuelve el resto de una división. (Esta operación también se conoce como "módulo")	`5 % 4 = 1`
`+` (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`

Operador de Concatenación

⬆ El operador "+" puede usarse en un contexto especial con Textos para concatenarlos. Esto devuelve la unión de los dos Textos operandos en uno solo. Puedes leer más aquí.

```
ENVIAR "Quiero " + "café"   COMENTAR "Envía 'Quiero café'"
```

La sentencia SUMAR también permite añadir un Texto a una variable más rápido:

```
CARGAR unTexto con "Quiero "

COMENTAR "Envía 'Quiero café'"
SUMAR unTexto con "café"
ENVIAR unTexto
```

Operador de Agrupación

⬆ Los operadores de agrupación `( )` manipulan la precedencia de evaluación dentro de una expresión. Echemos un vistazo al siguiente ejemplo:

```
CARGAR a con 1
CARGAR b con 2
CARGAR c con 3

COMENTAR "Precedencia habitual, en donde b * c se evalúa primero"
CARGAR d con a + b * c

COMENTAR "Precedencia habitual con paréntesis"
CARGAR e con a + (b * c)

COMENTAR "Precedencia manipulada. a + b se evalúa primero"
CARGAR f con (a + b) * c

COMENTAR "Envía '7 7 9'"
ENVIAR Texto d, " ", e, " ", f
```

Eso es todo lo fundamental sobre operadores y expresiones. Si quieres aprender más sobre operadores, puedes revisar su sección en el capítulo de Referencia.

A continuación, veremos cómo darles vida a tus Tubérculos (y no hablo de jardinería).

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 ingresa cuando quiere ejecutar tu Tubérculo, con tal de realizar una acción diferente. 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 pasarán al Tubérculo.

Previamente explicamos la sentencia REGISTRAR para crear funciones. Ahora, veremos cómo usarla para registrar Entradas de Usuario. Volviendo al ejemplo:

```
REGISTRAR Entrada a con 2
REGISTRAR Entrada b con 2

ENVIAR "Resultado: ", a + b
```

REGISTRAR usado de esta forma puede ser confuso para principiantes por la forma en la que funciona. Antes de explicarlo a detalle, debemos comprender algo.

Primera Ejecución

Recuerda que cuando crees un Tubérculo con Bot, este hará una ejecución de prueba antes de guardarse. A esta se le llama "primera ejecución".

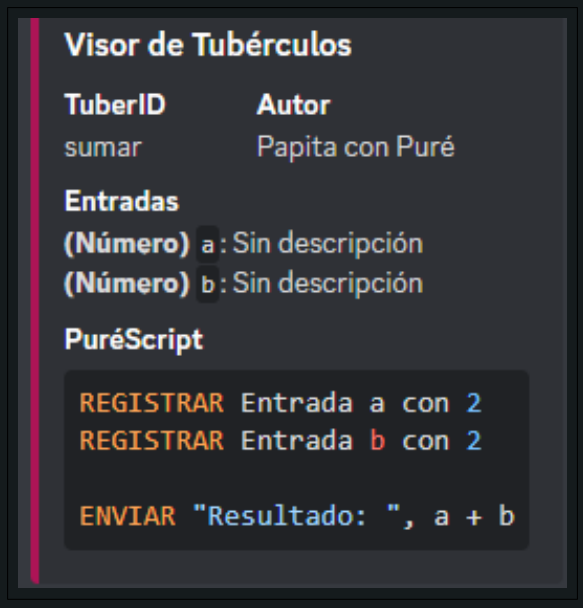
`REGISTRAR Entrada` se ignora por completo cuando un Usuario ejecuta el Tubérculo, pero tiene un comportamiento especial en la primera ejecución. En este contexto, la sentencia funciona similar a un CARGAR, asignándole un valor al identificador especificado.

En el ejemplo anterior, podemos pensar en los `con 2` como "valores de prueba" para las entradas `a` y `b`. La primera ejecución de dicho Tubérculo enviaría "Resultado: 4".

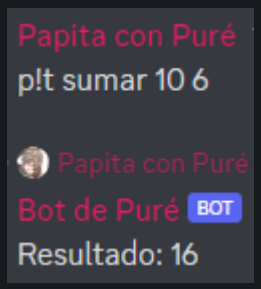
Ejecuciones Posteriores

En ejecuciones posteriores, el "valor de prueba" será reemplazado por un dato que escriba el Usuario al ejecutar el Tubérculo (o sea, se guardará la Entrada en el Tubérculo).

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. Sin embargo, los valores `con 2` que le asignamos a cada una ahora serán ingresados por cualquier Usuario que ejecute el Tubérculo.



Nótese que el tipo de valor se conserva, y es el único tipo de valor que tiene permitido ingresar el Usuario, de lo contrario, se alza un error.



Aquí, como podemos ver, al ingresar los valores "10" y "6", son transformados a Números y luego sumados para enviar el resultado de la suma. Es importante mencionar que **los valores se ingresan en el orden que se registraron**, o sea que `10` se le asigna a la variable `a` y luego `6` se le asigna a la variable `b`.

Con esto en mente, podemos registrar Entradas de múltiples tipos. El tipo de la Entrada se determina con el tipo del "valor de prueba" que escribes en el código, así que recuerda asignar un buen valor de prueba en el código. Si quieres recibir un Texto pero ingresas un Número de valor de prueba, vas a recibir un Texto, que puede o no ser convertible a Número. Ten cuidado con eso.

Hablando de tipos, es importante mencionar que no puedes registrar entradas de Listas y Glosarios, ya que sería muy complicado e incluso confuso de ingresar para el Usuario.

Los tipos de Entrada que puedes registrar son Números, Textos y Duplas. Siempre se hará cumplir el ingreso correcto de estos datos, así que no tienes que preocuparte por que la Entrada no exista o sea de un tipo diferente, solo por su valor.

Ingreso de Diferentes Tipos

Puede que recuerdes que vimos muy brevemente el `REGISTRAR Entrada` con un ejemplo de funciones. En ese caso, estábamos registrando una entrada de Texto:

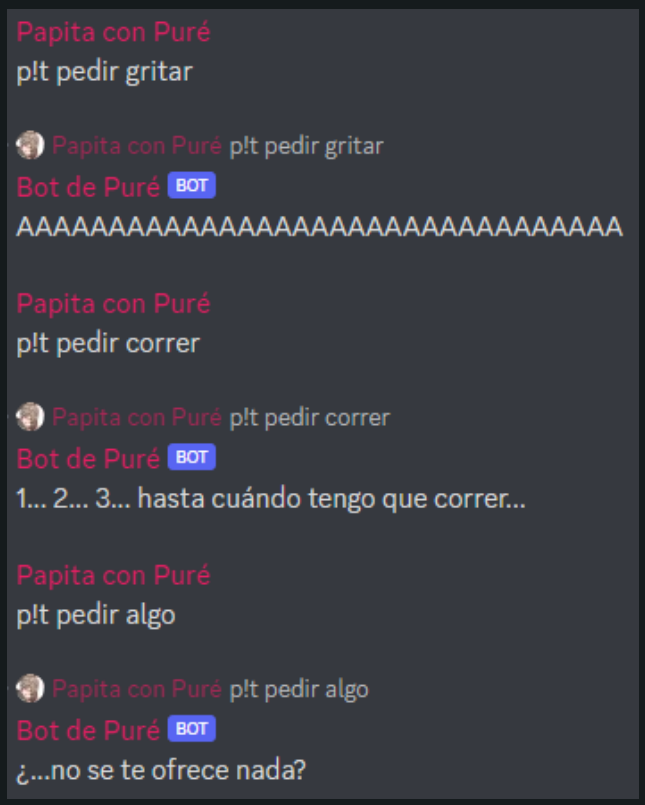
```
REGISTRAR Entrada acción con "gritar"

CREAR Función hacerAlgo

SI acción es "gritar"
    CARGAR hacerAlgo con Función () BLOQUE
        ENVIAR "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
    FIN
SINO SI acción es "correr"
    CARGAR hacerAlgo con Función () BLOQUE
        ENVIAR "1... 2... 3... hasta cuándo tengo que correr..."
    FIN
SINO
    CARGAR hacerAlgo con Función () BLOQUE
        ENVIAR "¿...no se te ofrece nada?"
    FIN
FIN

EJECUTAR hacerAlgo
```

Lo cuál se ejecutaría así:



Esto no involucra ninguna conversión. Es solo texto pasado directamente la 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.

En el caso de registrar Duplas, el Usuario debe ingresar Verdadero o Falso.

Ingresar cualquier otra cosa alzará un error.

Puedes ingresar "verdadero" o "falso" con cualquier minúscula y mayúscula que quieras.

Como podrás imaginar, el Registro de Entradas de tipo Dupla se hace asignando un valor de prueba de `Verdadero` o `Falso`.

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

Puede ser complicado de entender al principio, pero el registro de Entradas de Usuario debería ser bastante sencillo y útil.

Explicado de otra forma, se puede dividir el proceso en 3 partes:

1. El programador especifica una Entrada `var` y un valor de prueba para esta.

2. El Tubérculo realiza la primera ejecución cargando a `var` con el valor de prueba.

3. El valor de prueba se descarta y posteriores ejecuciones del Tubérculo cargarán `var` con un valor ingresado por el Usuario que ejecuta el Tubérculo.

Trabajando con Marcos

Ya vimos casi todo lo básico en PuréScript, pero todavía no hemos indagado en qué son los Marcos ni cómo utilizarlos. En esta sección aprenderemos al respecto.

Por empezar, en caso de que no sepas a qué nos referimos con "Marco", mira a tu derecha:

Ahora... ¿quieres aprender cómo crear uno?

Quieres hacerlo, ¿no? Se ve cool, ¿verdad? ¿¡No es así!?

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 Marco vacío por defecto, lo cual nos permite modificarlos con unas funciones nativas especiales para la manipulación de Marcos:

```
CREAR Marco unMarco

EJECUTAR marcoAsignarTítulo(unMarco, "Esto es un Marco")
EJECUTAR marcoAsignarColor(unMarco, colorRojo)

COMENTAR "Buscamos a Bot de Puré por medio de su ID..."
CARGAR bot con buscarMiembro("651250669390528561")
EJECUTAR marcoAsignarImagen(unMarco, bot->avatar)

EJECUTAR marcoAgregarCampo(unMarco, "Mucho gusto", "Encantado de conocerte")

ENVIAR unMarco
```

Nótese el uso de paso por referencia para modificar la variable `unMarco`. Los Marcos son como una extensión de los Glosarios, así que también se pasan por referencia a funciones.

(La función `buscarMiembro` simplemente busca un miembro y lo representa en un Glosario).

Funciones de Marco

No te asustes por la cantidad de funciones que muestra el ejemplo, están todas listadas y explicadas en la Referencia. De todas formas, hay que explicar brevemente algunas de ellas.

Las funciones de Marco son funciones nativas, así que las puedes acceder desde cualquier parte. Nótese el uso de `Asignar` y `Agregar` en los nombres de las funciones, ya que describen cómo se relacionan los datos al Marco. "Asignar" hace referencia a "añadir si no existe o reemplazar si ya existe", mientras que "Agregar" solo agrega.

Estas son 3 funciones de Marco bastante comunes:

`marcoAsignarTítulo(marco, título)`

> Asigna un `título` al `marco`.

`marcoAsignarColor(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.

`marcoAgregarCampo(marco, nombre, valor)`

> Añade un campo al `marco`. Dicho campo lleva un `nombre` (algo similar a un título) y un `valor` que va justo debajo del nombre. Según las reglas de Discord, puedes ingresar 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.

¡Este es el final de la Guía por ahora!

A continuación, podrías intentar programar tus propios Tubérculos y consultar la Referencia en caso de tener problemas.

¡Suerte!

Ilustración por Rakkidei.

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, te recomiendo ver la Guía.

Los campos de Sintaxis harán uso de "<>" para ejemplificar campos que se supone que rellene el Programador al crear un Tubérculo.

Índice de Capítulo

Ilustración https://twitter.com/pu_yan8901

REGISTRAR

Indicador de Sentencia

Sintaxis

```
REGISTRAR <Tipo> <Identificador> con <Expresión>
```

Opera de diferentes formas dependiendo del `<Tipo>` de registro ingresado.

Campo	Opcional	Descripción
Tipo		Tipo de registro del identificador.
Identificador		Nombre bajo el cual se referirá a lo registrado.
Expresión		Expresión que se le asignará al identificador.

Tipos de Registro

Registro de Entrada

Registra una Entrada de Usuario en el Tubérculo como una variable bajo el `<Identificador>` ingresado. En la ejecución de prueba (cuando se está por crear el Tubérculo), en lugar de usar una entrada de Usuario, se usará la `<Expresión>` proporcionada, conocida como el "valor de prueba". El tipo de la expresión definirá el tipo de la variable declarada. Esta sentencia se ignora luego de la primera ejecución, y queda en cambio grabada en las entradas del Tubérculo para el ingreso del Usuario.

```
REGISTRAR Entrada <Identificador> con <Expresión>
```

Es importante tener en cuenta que las entradas se deben registrar en el orden que quieras que las ingrese el Usuario. La primer entrada que declares, siempre será la primera que ingrese el Usuario; la segunda que declares, la segunda que ingrese; y así.

Registro de Lista

Registra una Lista bajo el `<Identificador>` ingresado, con los valores extraídos del archivo en la `<Expresión>`. Es una alternativa a la sentencia CARGAR si la Lista es muy larga. Este tipo de registro usa la Lista especial "archivos" para cargar un archivo enviado en el mensaje del Programador.

Registro de Función

Registra una Función bajo el `<Identificador>` especificado. La `<Expresión>` debe ser un bloque representando el cuerpo de la Función.

El cuerpo de una Función es básicamente un bloque que se ejecuta solo cuando la Función es llamada en otra parte del Programa, por medio del identificador especificado.

Este tipo de registro difiere del resto en el hecho de que requiere paréntesis luego del identificador, dentro de los cuáles se pueden describir los argumentos que recibirá la Función, separados por coma. Esta es la sintaxis básica:

```
REGISTRAR Función fn(argumentos...) con BLOQUE
    ...sentencias
FIN
```

Nota: las Funciones en PuréScript deben ser registradas/declaradas antes de ser utilizadas.

CREAR

Indicador de Sentencia

Sintaxis

```
CREAR <Tipo> <Identificador>
```

Crea una variable bajo el `<Identificador>` proporcionado, del `<Tipo>` especificado. La variable se inicializa en el valor por defecto del tipo de variable, si tiene.

Campo	Opcional	Descripción
Tipo		Tipo de la variable a declarar.
Identificador		Nombre bajo el cual se declara la variable.

Valores por defecto

- Número: `Nada`.
- Texto: `Nada`.
- Dupla: `Falso`.
- Lista: `<Lista vacía>`.
- Glosario: `<Glosario vacío>`.
- Marco: `<Marco vacío>`.

GUARDAR

Indicador de Sentencia

Sintaxis

```
GUARDAR <Identificador> con <Expresión>
```

La sentencia GUARDAR no está disponible en esta versión de PuréScript.

Campo	Opcional	Descripción
Identificador		Nombre bajo el cual guardar el dato.
Expresión		Expresión a guardar.

¡Advertencia!
Esta sentencia aun no está implementada y puede ser eliminada o cambiada en el futuro. Tanto en sintaxis como en comportamiento.

CARGAR

Indicador de Sentencia

Sintaxis

```
CARGAR <Identificador> con <Expresión>
```

Le 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.

Declaración Rápida

Si no existe ninguna variable bajo dicho identificador, se la declara bajo el tipo y valor de la expresión. Lo cuál muestra un comportamiento idéntico a:

```
CREAR <Tipo> <Identificador>
CARGAR <Identificador> con <Expresión>
```

Como se ve, la secuencia CARGAR puede actuar también como la secuencia CREAR. Sin embargo, es importante mencionar que la secuencia CREAR existe por sus propios motivos particulares. Si bien suele ser usada para declarar listas y glosarios y luego extenderlos, también puede usarse en casos que quieras asignarles valores a variables dentro de bloques y conservar el valor al finalizar el mismo. Esto se logra declarando la variable fuera del bloque con la secuencia CREAR.

Ejemplos

El siguiente programa tiene una probabilidad de 10% de enviar un mensaje sobre el otro:

```
CARGAR aleatorio con dado(1)

CREAR Texto mensaje
SI aleatorio 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 dependiendo de la cantidad especificada por el Usuario:

```
REGISTRAR Entrada cantidad con 3

SI cantidad es 0
    ENVIAR "La canasta no tiene frutas :("
    TERMINAR
FIN

CREAR Lista canasta
MIENTRAS cantidad excede 0 RESTAR cantidad
    COMENTAR "Se declara la lista 'frutas' en cada iteración"
    CARGAR frutas con Lista
        "Manzana",
        "Naranja",
        "Banana",
        "Pera",
    CARGAR aleatorio con dado(frutas->largo, Verdadero)

    EXTENDER canasta con frutas->aleatorio
FIN

COMENTAR "La Lista 'frutas' ya no existe en este punto del código,
o sea que se puede declarar una nueva variable 'frutas'"
CREAR Texto frutas
CARGAR frutas con canasta->unir(", ")
ENVIAR frutas
```

SUMAR

Indicador de Sentencia

Sintaxis

```
SUMAR <Identificador>
SUMAR <Identificador> con <Expresión>
```

Suma un valor a la variable bajo el `<Identificador>` especificado.

Campo	Opcional	Descripción
Identificador		Nombre de una variable.
Expresión	✔[1]	Expresión a sumarle a la variable.

Sin Expresión

Similar comportamiento a `CARGAR <Identificador> con <Identificador> + 1`, o sea que se le suma 1 a la variable y se guarda el resultado en la misma. La diferencia con CARGAR es que no ocurre ninguna Declaración Rápida. La variable debe ser un Número.

Con Expresión

Similar comportamiento a `CARGAR <Identificador> con <Identificador> + <Expresión>`, o sea que se suman la variable y la expresión, y se guarda el resultado de la operación en la misma. La diferencia con CARGAR es que no ocurre ninguna Declaración Rápida.

La variable puede ser un Número o un Texto. Si se le suma a una variable de tipo Texto, se tomará como un caso de concatenación entre los contenidos de la variable y la expresión. No se puede sumar a Textos sin describir una expresión.

1 Solo es opcional si la variable mencionada es un Número.

RESTAR

Indicador de Sentencia

Sintaxis

```
RESTAR <Identificador>
RESTAR <Identificador> con <Expresión>
```

Resta un Número a la variable numérica bajo el `<Identificador>` especificado.

Campo	Opcional	Descripción
Identificador		Nombre de una variable.
Expresión	✔	Expresión a restarle a la variable.

Sin Expresión

Similar comportamiento a `CARGAR <Identificador> con <Identificador> - 1`, o sea que se le resta 1 a la variable y se guarda el resultado en la misma. La diferencia con CARGAR es que no ocurre ninguna Declaración Rápida.

Con Expresión

Similar comportamiento a `CARGAR <Identificador> con <Identificador> - <Expresión>`, o sea que se resta la expresión a la variable y se guarda el resultado de la operación en la misma. La diferencia con CARGAR es que no ocurre ninguna Declaración Rápida.

MULTIPLICAR

Indicador de Sentencia

Sintaxis

```
MULTIPLICAR <Identificador> con <Expresión>
```

Multiplica la variable numérica bajo el `<Identificador>` mencionado con el Número de la `<Expresión>` especificada.

Campo	Opcional	Descripción
Identificador		Nombre de una variable.
Expresión		Número por el cual multiplicar la variable.

Similar comportamiento a `CARGAR <Identificador> con <Identificador> * <Expresión>`, o sea que se multiplica la variable por la expresión y se guarda el resultado de la operación en la misma. La diferencia con CARGAR es que no ocurre ninguna Declaración Rápida.

DIVIDIR

Indicador de Sentencia

Sintaxis

```
DIVIDIR <Identificador> con <Expresión>
```

Divide la variable numérica bajo el `<Identificador>` mencionado por el Número de la `<Expresión>` especificada.

Campo	Opcional	Descripción
Identificador		Nombre de una variable.
Expresión		Expresión por la cual dividir la variable.

Similar comportamiento a `CARGAR <Identificador> con <Identificador> / <Expresión>`, o sea que se divide la variable por la expresión y se guarda el resultado de la operación en la misma. La diferencia con CARGAR es que no ocurre ninguna Declaración Rápida.

EXTENDER

Indicador de Sentencia

Sintaxis

```
EXTENDER <Identificador> con <Expresión>
EXTENDER <Identificador>
      con <Expresión>
      en <Miembro>
```

Añade un elemento a la Lista bajo el `<Identificador>` indicado.

Campo	Opcional	Descripción
Identificador		Nombre de una Lista.
Expresión		Expresión a asignar al elemento mencionado.
Miembro	✔	Nombre de un elemento de la Lista.

Expresión

La expresión puede ser cualquier valor, incluso una nueva Lista dentro de la Lista a extender, u otra estructura compleja.

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 se encuentra dentro del largo de la Lista, primero se empujan por +1 índice todos los elementos desde el índice especificado hasta el final de la lista.

- Si el índice se encuentra más allá del largo de la lista, se añaden los elementos vacíos necesarios para insertar el nuevo elemento.

EJECUTAR

Indicador de Sentencia

Sintaxis

```
EJECUTAR <Identificador>()
EJECUTAR <Identificador>(<Argumentos...>)
EJECUTAR <Tipo> <Identificador>(<Argumentos...>)
```

Ejecuta una Función, un comando o un Tubérculo. La ejecución de comandos y Tubérculos deben especificar el `<Tipo>` de ejecución. Los envíos de mensaje desencadenados en la ejecución de comandos o Tubérculos serán añadidos al envío del Tubérculo que la realiza.

Nótese que los `<Argumentos...>` pueden ser opcionales y varían en tipo y cantidad de Función en Función.

Campo	Opcional	Descripción
Tipo	✔[2]	Tipo de ejecución en relación al identificador.
Identificador		Nombre de función, comando o Tubérculo.
Argumentos...	✔	Secuencia de valores a pasar a la ejecución.

Tipos de Ejecución

El concepto de ejecutar no varía mucho entre cada tipo. En el caso de funciones, no se agrega ningún envío de mensaje al Tubérculo que llama, pero en el caso de ejecutar comandos y Tubérculos, se añaden sus envíos de mensaje al envío de mensaje del Tubérculo que llama. Nótese que la adición de envío de los comandos y Tubérculos solo funciona con esta sentencia.

2 Solo es opcional si se trata de una ejecución de función.

Función

Ejecuta una función bajo el `<Identificador>` indicado, con los `<Argumentos...>` expresados.

```
COMENTAR "Asigna un encabezado y campo a un Marco, y lo envía"

CREAR Marco miMarco
EJECUTAR marcoAsignarEncabezado(miMarco, "Un Buen Encabezado")
EJECUTAR marcoAgregarCampo(miMarco, "Título", "Párrafo...")
ENVIAR miMarco
```

Comando

> **¡Advertencia!**
> Esta característica aun no está implementada y puede cambiar en el futuro.

Ejecuta un comando bajo el `<Identificador>` indicado, con los `<Argumentos>` expresados. El mensaje que enviaría el comando se añade a los envíos del Tubérculo que lo invoca.

Cabe mencionar que, incluso si el medio de acceso al comando es un identificador, los comandos en sí mismos no distinguen mayúsculas, así que puedes usar cualquier nombre o alias de comando mientras no te equivoques con las tildes.

Ejecutar un comando de esta forma sigue requiriendo que el Usuario tenga los permisos necesarios, y que el comando esté habilitado en el servidor.

```
COMENTAR "Replica un 'p!papita <frase>', con un texto añadido por encima"

REGISTRAR Entrada puré con "Quiero un café"
ENVIAR "🥔 dice:"
EJECUTAR Comando papita(puré)
```

Es de notable mención que, si bien la mayoría de comandos solo responden con 1 mensaje, existen otros comandos que envían varios mensajes en secuencia. En el caso de estos comandos, solo se añadirá al envío el primer mensaje.

Tubérculo

> **¡Advertencia!**
> Esta característica aun no está implementada y puede cambiar en el futuro.

Ejecuta un Tubérculo bajo el `<Identificador>` indicado, con los `<Argumentos>` expresados. El mensaje que enviaría el comando se añade a los envíos del Tubérculo que lo invoca.

Ahora un ejemplo. En un servidor existen los Tubérculos "papainador" y "batatainador":

```
REGISTRAR Entrada persona con "bot"
CARGAR persona con buscarMiembro(persona)
ENVIAR "🥔 ", persona, " sos una papa 🥔"
```

```
REGISTRAR Entrada persona con "bot"
CARGAR persona con buscarMiembro(persona)
ENVIAR "🍠 ", persona, " es una batata 🍠"
```

Si queremos crear un Tubérculo "tubérculoinador" que una los envíos de ambos:

```
REGISTRAR Entrada búsqueda con "bot"
EJECUTAR Tubérculo papainador(búsqueda)
EJECUTAR Tubérculo batatainador(búsqueda)
```

En el caso anterior, se ejecutan ambos Tubérculos en orden, con el argumento `búsqueda`. Dicho argumento se interpreta como si fuera una entrada de Usuario en el Tubérculo llamado. O sea, como si los llamaras con "p!tubérculo papainador `búsqueda`", por ejemplo.

DEVOLVER

Indicador de Sentencia

Sintaxis

```
DEVOLVER <Expresión>
```

Termina la ejecución de la Función actual y devuelve la `<Expresión>` ingresada a donde se llamó la función.

Campo	Opcional	Descripción
Expresión		Valor a devolver.

Devolución de Valores en Funciones

Todas las Funciones pueden devolver un valor. Ya sea este un Texto, un Glosario, etc. Cualquier Función que no ejecute esta sentencia devolverá `Nada` por defecto.

En caso de solo querer terminar la ejecución de la Función sin la necesidad de devolver un valor, puedes simplemente devolver `Nada`, o usar TERMINAR.

```
DEVOLVER Nada
```

Omisión de Sentencias

DEVOLVER omite el resto de sentencias hacia abajo en el cuerpo de la Función. No puede ser usado directamente en el bloque Programa.

Todas las sentencias que se estén en el mismo ámbito y *luego* de esta sentencia serán ignoradas directamente por el analizador.

TERMINAR

Indicador de Sentencia

Sintaxis

```
TERMINAR
```

Termina la ejecución del bloque iterativo (MIENTRAS, PARA, etc) o cuerpo de función actual. Si no se está dentro de ninguno de estos, termina el Bloque Programa.

Omisión de Sentencias

TERMINAR omite el resto de sentencias hacia abajo en el bloque iterativo / cuerpo de Función actual.

Todas las sentencias que se estén dentro del mismo ámbito y *luego* de esta sentencia serán ignoradas directamente por el analizador.

Si TERMINAR no se usa *directamente dentro* de un bloque iterativo o cuerpo de función, se traspasarán **todas** las sentencias posteriores al bloque actual hasta llegar al fin de uno de estos 2, o el fin del bloque Programa.

BLOQUE

Indicador de Sentencia » Bloque

Sintaxis

```
BLOQUE
    ...sentencias a ejecutar dentro del bloque
FIN
```

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. Pasa señalizar el final de un bloque, se usa FIN.

Todas las sentencias de Bloque siguen una extensión de este comportamiento. Incluyendo el terminar con FIN.

FIN

Indicador de Sentencia » Bloque

Sintaxis

```
BLOQUE
    ...sentencias
FIN
```

Termina el bloque actual, destruyendo su ámbito de bloque 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 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.

SI

Indicador de Sentencia » Bloque » Estructura de Control Condicional

Sintaxis

```
SI <Expresión>
    ...sentencias a ejecutar si la expresión es verdadera
FIN
```

Marca el inicio de un bloque que se ejecuta si la `<Expresión>` puede evaluar a `Verdadero`.

Campo	Opcional	Descripción
Expresión		Expresión a evaluar para controlar la ejecución u omisión del bloque contenido.

Evaluación Condicional

Expresión Lógica

Se suelen usar expresiones lógicas como evaluación condicional.

```
SI a y b
```

Expresión Comparativa

También es común usar expresiones comparativas.

```
SI a es b
```

Expresiones Varias

También puedes usar expresiones complejas y simples valores.

```
SI Verdadero
SI "¡Hola!"
SI 42
SI unValor
SI a * b - c
SI (a es b o b excede c) y a + b precede d
```

SINO

Indicador de Sentencia » Bloque » Estructura de Control Condicional

Sintaxis

```
SI <Expresión>
    ...sentencias a ejecutar si la expresión es verdadera
SINO
    ...sentencias a ejecutar si la expresión es falsa
FIN
```

Es la combinación de las palabras "si no" para formar un indicador de sentencia.

Marca el inicio de un bloque inmediatamente después de un bloque SI. Dicho bloque solo se ejecuta si el bloque SI previo no fue ejecutado (*si no* se ejecuta).

Funcionalidad Condicional

Este bloque solo se ejecuta si el bloque SI previo no fue ejecutado. Lo cual significa que, si el bloque previo se ejecuta, este se ignorará.

ESTRUCTURA SINO...SI

Puedes concatenar múltiples comprobaciones lógicas de la siguiente forma:

```
SI a
    COMENTAR "Esto se ejecuta si 'a' es Verdadero"
SINO SI b
    COMENTAR "Esto se ejecuta si 'a' es Falso y 'b' es Verdadero"
SINO
    COMENTAR "Esto se ejecuta si 'a' y 'b' evalúan a Falso"
FIN
```

MIENTRAS

Indicador de Sentencia » Bloque » Estructura de Control Iterativa

Sintaxis

```
MIENTRAS <Expresión>
    ...sentencias a ejecutar si la expresión es verdadera
FIN
```

Marca el inicio de un bloque que se ejecuta mientras la `<Expresión>` evalúa a `Verdadero`.

Campo	Opcional	Descripción
Expresión		Expresión a evaluar cada vez que se intenta iterar el bloque contenido.

MIENTRAS comparte características de las sentencias BLOQUE y SI.

En lugar de ejecutar el bloque que contiene y seguir con las sentencias debajo, esta sentencia repite el bloque una y otra vez mientras la condición siga evaluando a `Verdadero`.

Bucles Infinitos

Si la expresión del MIENTRAS nunca evalúa a `Falso`, ocurrirá un bucle infinito, lo cuál eventualmente causará un error por límite de sentencias procesadas. Siempre vas a querer asegurarte de que los contenidos del bloque iterativo harán que eventualmente la expresión evalúe a `Falso`. Puedes hacer eso o usar TERMINAR en el bloque para cortar las iteraciones posteriores e interrumpir la iteración actual.

HACER YSEGUIR MIENTRAS

Indicadores de Sentencia » Bloque » Estructura de Control Iterativa

Sintaxis

```
HACER
    ...sentencias a ejecutar 1 vez y luego si la expresión es verdadera
YSEGUIR MIENTRAS <Expresión>
```

HACER marca el inicio de un bloque que se ejecuta mientras la `<Expresión>` evalúa a `Verdadero`. A diferencia de MIENTRAS, el bloque se ejecuta al menos una vez, incluso si la Expresión nunca se evalúa como `Verdadero`.

Este es el único bloque que no finaliza con FIN, sino que hace uso de YSEGUIR.
YSEGUIR como tal debe ser acompañado de un MIENTRAS con la `<Expresión>` del bloque.

Campo	Opcional	Descripción
Expresión		Expresión a evaluar cada vez que se intenta iterar el bloque contenido luego de la primer iteración.

HACER YSEGUIR MIENTRAS tiene un comportamiento muy similar a MIENTRAS. La única diferencia es que primero se ejecuta el bloque y luego se comprueba la expresión a cumplir para iterar nuevamente. Esto hace que el bloque se ejecute al menos una vez.

Bucles Infinitos

Si la expresión nunca evalúa a `Falso`, ocurrirá un bucle infinito, lo cuál eventualmente causará un error por límite de sentencias procesadas. Siempre vas a querer asegurarte de que los contenidos del bloque iterativo harán que eventualmente la expresión evalúe a `Falso`. Puedes hacer eso o usar TERMINAR dentro del mismo para cortar las iteraciones posteriores e interrumpir la iteración actual.

PARA

Indicador de Sentencia » Bloque » Estructura de Control Iterativa

Sintaxis

```
PARA <Asignación> MIENTRAS <Expresión> <SUMAR/RESTAR> <Identificador>
    ...sentencias a ejecutar si la expresión es verdadera
FIN
```

Marca el inicio de un bloque que se ejecuta mientras la `<Expresión>` evalúa a `Verdadero`.

Asignación de Variable Contador

PARA crea un ámbito en el cual realiza la `<Asignación>` especificada, para luego iterar sobre el bloque contenido. La asignación declarará una variable contador bajo en identificador que quieras. El bloque de la estructura como tal recibe un nuevo ámbito anidado en cada iteración, como ocurre con el resto de estructuras iterativas.

Actualización (Incremento/Decremento)

Al final de cada iteración, se suma o resta a la variable del `<Identificador>` dependiendo del indicador de sentencia previo. El identificador en la asignación debería idealmente ser el mismo que el que se actualiza en cada iteración.

Campo	Opcional	Descripción
Asignación		Expresión en la cual declaras el `<Identificador>` y le asignas un valor inicial, antes de comenzar a iterar.
Expresión		Expresión a comprobar cada vez que se intenta iterar sobre el bloque contenido.
Identificador		El identificador a actualizar al final de cada iteración sobre el bloque contenido.

PARA imita el comportamiento de un MIENTRAS junto a una variable contador:

```
CARGAR i con 0
MIENTRAS i precede 42 SUMAR i
    ...sentencias a ejecutar si la expresión es verdadera
FIN
```

La única diferencia es que la variable contador es declarada "dentro del ámbito de la sentencia PARA", por lo cuál esta será destruída al finalizar la ejecución de las iteraciones:

```
CARGAR i con 0
MIENTRAS i precede 42
    COMENTAR "Esto se ejecuta 42 veces con diferentes valores de 'i'"
    SUMAR i
FIN

COMENTAR "Envía Falso y 42"
ENVIAR esNada(i)
ENVIAR i
```

```
PARA i con 0 MIENTRAS i precede 42 SUMAR i
    COMENTAR "Esto se ejecuta 42 veces con diferentes valores de 'i'"
FIN

COMENTAR "Envía Verdadero y 'Nada'"
ENVIAR esNada(i)
ENVIAR Texto i
```

PARA CADA

Puedes usar el indicador CADA inmediatamente después del indicador PARA si quieres iterar sobre una lista. Esto cambia la sintaxis y hace uso de la palabra clave "en":

```
PARA CADA <Identificador> en <Lista>
    ...esto se ejecuta una vez por cada elemento dentro de la <Lista>.
    <Identificador> se referirá a dicho elemento en cada iteración
FIN
```

Esta estructura es útil cuando queremos iterar sobre una Lista y no nos importa saber el índice en cada iteración. Nótese que no tendrás acceso al índice de la iteración actual con esta estructura. La ventaja es que cualquier operación con la Lista iterada se vuelve bastante más compacta. Por ejemplo:

```
CARGAR unaLista con Lista 0, 1, 2

COMENTAR "Envía el valor de todos los elementos con un formato"
PARA CADA elemento en unaLista
    ENVIAR "Este elemento vale: " + elemento
FIN
```

En este caso, `elemento` representa el valor del elemento de `unaLista` que está siendo iterado actualmente. El identificador que se crea en cada iteración representa una copia del valor más que una referencia al mismo, así que modificarlo no reflejará los cambios en la Lista original. Para eso, tendrás que sí o sí usar el PARA común.

```
CARGAR unaLista con Lista 0, 1, 2
PARA CADA elemento en unaLista
    CARGAR elemento con elemento * 2
FIN
COMENTAR "Envía '0, 1, 2'"
ENVIAR unaLista->unir(", ")
```

ENVIAR

Indicador de Sentencia

Sintaxis

```
ENVIAR <Expresión>
```

Envía un mensaje en Discord que corresponde a la `<Expresión>` ingresada. En caso de ser una ejecución de un Tubérculo dentro de otro, envía el valor de la expresión al Tubérculo que lo llama.

Campo	Opcional	Descripción
Expresión		Contenido a enviar en el mensaje.

El mensaje no se envía en el momento, 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 ENVIAR añade un renglón al mensaje enviado.

Tipos de Envío

Envío de Texto

Si la expresión es de tipo Texto, se enviará como texto en el mensaje.

```
ENVIAR "Esto es un mensaje normalito, común, tradicional, etc."
ENVIAR "Esto es... ¿otro mensaje? No realmente"
ENVIAR "Hey ", usuario, ", bonita pfp"
```

Enviar múltiples Textos hace que se pongan uno debajo del otro, en orden de envío.

Envío de Enlaces de Texto

Si la expresión es de tipo Texto y el contenido se detecta como un enlace, se intentará enviar como un archivo adjunto al mensaje. Requiere el protocolo HTTP/S al inicio.

```
CREAR Lista imágenes
ENVIAR "https://ejemplo.com/imagen.png"
ENVIAR "https://e.xample.net/p2.png"
```

Envío de Marco

Si la expresión es de tipo Marco, se enviará como un marco en el mensaje.

```
CREAR Marco mensaje
CARGAR título con "¡Buen estilo!"
CARGAR frase con "Este tipo de mensajes suele verse bien."
EJECUTAR marcoAgregarCampo(mensaje, título, frase)
ENVIAR mensaje
```

Envío de Convertible a Texto

No se puede enviar `Nada` sin convertirlo a Texto. Si lo haces, se alzará un error.

```
ENVIAR Nada
ENVIAR variableInexistente
```

Si la expresión no entra en ninguna de las anteriores categorías, se intentará convertir a Texto antes de enviarse. Si esto falla, se alzará un error.

```
ENVIAR Texto variableInexistente    COMENTAR "Envía 'Nada'"
ENVIAR Lista 1, 2, "hola", 3, Falso COMENTAR "Envía '12hola3Falso'"
ENVIAR Glosario a: 1, b: "2", c: 3  COMENTAR "Envía 'a: 1, b: 2, c: 3"
```

COMENTAR

Indicador de Sentencia

Sintaxis

```
COMENTAR <Texto>
```

Permite introducir un texto a modo de comentario.

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"
```

Número

Tipo de dato primitivo de variable

Representa un valor numérico en el dominio de los números reales. Comúnmente devuelto por operaciones aritméticas.

Expresión Literal

Usando el valor numérico 0 de referencia:

```
0
Número 0
```

Valor por Defecto

`Nada`.

Ejemplos

- `0`
- `42`
- `-74`
- `3.333333333333333`

Conversiones

- A Texto: `Texto 3` = “3”.
- A Dupla: `Falso` si el número es 0; `Verdadero` en cualquier otro caso.

Revisa los Métodos de Número

Puedes ver todos los métodos de Número aquí.

Indicador de Tipo de Número

El indicador de tipo es opcional para los Números. Sin embargo, puede añadirse como azúcar sintáctica.

Operaciones de Números

Puedes usar los operadores matemáticos (“+”, “-”, “*”, etc) para realizar cálculos entre Números. Así mismo, puedes operar entre Números y otros tipos de expresiones que puedan convertirse en Números. En caso de múltiples operaciones en sucesión, se aplica precedencia de operadores. Lo cuál significa que por ejemplo “4 + 3 * 2” se calcula como “4 + (3 * 2)” en lugar de “(4 + 3) * 2”. Operadores de misma precedencia se calculan de izquierda a derecha, como en el caso de una sucesión de sumas y restas “1 + 2 - 3 + 4”

```
CARGAR tremendaFórmula con 2 + 4 * (pi * 2 - 1) / 5 - 2 * 0.5 + 3
ENVIAR "No sé por qué harías esto, pero... ", tremendaFórmula
```

Operaciones entre Expresiones

Por ejemplo, si realizas un cálculo entre un Número y una Dupla, la Dupla se convertirá a `1` si es `Verdadero` o `0` si es `Falso` y luego se operará con el Número.

Texto

Tipo de dato primitivo de variable

Representa un valor textual. Puede originarse de una plantilla de Texto.

Expresión Literal

Usando el valor textual “puré de papa” de referencia:

```
"puré de papa"
Texto "puré de papa"
```

Valor por Defecto

`Nada`.

Ejemplos

- `"¡Hola! ¿Andamos bien?"`
- `"42"`
- `"verdadero"`

Conversiones

- A Número: `Número "3"` = `3`; `Número "a"` = `0`.
- A Dupla: `Falso` si el texto está vacío; `Verdadero` en cualquier otro caso.

Todos los Textos tienen un miembro `->largo` indicando su cantidad de caracteres actual.

Concatenación de Texto

Puedes unir 2 Textos por medio del operador "+". También puedes unir Texto junto a cualquier expresión convertible en Texto (como un Número o Dupla).

```
CARGAR cantidad con 23423634235745
CARGAR frase con "Le puse " + cantidad + " papas al puré"
CARGAR comentario con cantidad + " papas fueron sacrificadas"
```

Plantillas de Texto

También puedes lograr un efecto similar con otro tipo de sintaxis en la cual las expresiones se separan con coma.

```
CARGAR puntos con guardado->puntosGuardados
CARGAR suma con dado(8, Verdadero)
SUMAR puntos con suma
GUARDAR puntosGuardados con puntos

ENVIAR Texto suma, " punto(s) obtenido(s)"
ENVIAR "¡Ahora tienes ", puntos, " puntos!"
```

Nótese cómo, cuando la plantilla empieza con un literal de texto, se puede omitir el indicador de tipo. Sin embargo, cuando empieza con un identificador, Número u otra cosa, debes de agregarlo o no se detectará como una plantilla.

Revisa los Métodos de Texto

Puedes ver todos los métodos de Texto desde aquí.

Dupla

Tipo de dato primitivo de variable

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.

Uso

Su utilización primaria es la *comprobación de bloques condicionales*, como lo serían los iniciados por las sentencias SI, MIENTRAS, PARA, etc.

Expresiones Literales

```
Verdadero
Falso
Dupla Verdadero
Dupla Falso
```

Valor por Defecto

`Falso`.

Ejemplos

- `Verdadero`
- `Falso`

Conversiones

- A Número: `Número Verdadero` = `1`; `Número Falso` = 0.
- A Texto: `Texto Falso` = `“Falso”`; `Texto Verdadero` = `”Verdadero”`.

Expresiones Lógicas

Así como puedes restar Números con "-" y concatenar Textos con "+", puedes relacionar 2 Duplas o expresiones con operadores lógicos.

Nótese que usar operadores aritméticos con Duplas las convertirá a Números para operarlas.

Algunos ejemplos de expresiones lógicas incluyen:

Operador "o"

Resulta `Verdadero` si al menos una de las expresiones es `Verdadero`. Si no, `Falso`.

Operador "y"

Resulta `Verdadero` si ambas expresiones son `Verdadero`. Si no, `Falso`.

Operador "es"

Resulta `Verdadero` si ambas expresiones valen lo mismo. Si no, `Falso`.

Operador "no"

Operador unario. Resulta `Falso` si la expresión es `Verdadero` y viceversa.

Precedencia de Operadores Lógicos

Al igual que con operaciones matemáticas, los operadores lógicos tienen un orden de evaluación según su precedencia:

"o" < "y" < "es"/"no es" < "parece"/"no parece" < "excede"/"precede" < "no" .

Puedes agrupar expresiones lógicas con paréntesis.

Lista

Tipo de dato estructural de variable

Representa un conjunto ordenado de valores/expresiones, denominados “elementos”. Puede incluso contener otras estructuras complejas.

Los elementos son indexados desde 0 en adelante. O sea que el primer elemento se encontrará en el índice 0 y el último en el índice del largo de la Lista menos 1.

Expresión Literal

Para expresar una Lista, se usa el indicador de tipo de Lista seguido de los valores/expresiones separados por coma.

```
Lista a, b, c
Lista d,
      e,
      f,
      g
```

Acceso a elementos con operador de flecha

Para acceder a los elementos de una Lista, se emplea el operador de flecha (“->”).

Si tenemos una lista `miLista` con X elementos y queremos acceder al elemento en el índice 3, hay que referenciar la lista y utilizar el operador de flecha junto al elemento de lista deseado. Resultando en la expresión “`miLista->3`”.

Intentar acceder un elemento que no existe en una lista devolverá `Nada`, y subsecuente uso del operador de flecha alzará un error.

Todas las Listas tienen un miembro `->largo` indicando su cantidad de elementos actual.

Valor por Defecto

Una lista vacía. O sea, sin elementos (y por lo tanto, sin índices accesibles).

Ejemplos

- `Lista 1, 2, 3`
- `Lista "hola, buenas tardes ", nombre, ". Son las ", 3`
- `Lista 42, unaVariable, unaFunción(), unMiembro->suPropiedad, Falso`

Conversiones

De lista sin elementos

- A Texto: `Texto <Lista vacía>` = `""`.
- A Dupla: `Dupla <Lista vacía>` = `Falso`.

De lista con elementos

- A Texto: `Texto (Lista "ho", "la")` = `"hola"`.
- A Dupla: `Dupla (Lista 1, 2, 3)` = `Verdadero`.

Coma Final Ignorable

Si pones una coma al final de un miembro, justo antes de un nuevo indicador de sentencia o fin de bloque, esta coma será ignorada. Esto es útil cuando programas y se ve mejor:

```
CARGAR li con Lista
    1,
    2,
    3,
    4,
    5,

COMENTAR "Envía '1, 2, 3, 4, 5'"
ENVIAR li->unir(", ")
```

Comas Extra

Si pones comas extra entre miembros, 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, , ,

COMENTAR "Envía '1.2.3.Nada.5.Nada.Nada'"
ENVIAR li->unir(".")
```

Revisa los Métodos de Lista

Puedes ver todos los métodos de Lista desde aquí.

Glosario

Tipo de dato estructural de variable

Representa un conjunto de pares relacionales de claves y valores, denominados “miembros”, similar a cómo un diccionario contiene pares relacionales de palabras y definiciones. Puede incluso contener otras estructuras complejas.

Los miembros de un glosario son asignados y accedidos con un identificador. A diferencia de las Listas, los miembros no se ordenan de ninguna forma particular.

Expresión Literal

Para expresar un Glosario, se usa el indicador de tipo de Glosario seguido de una sucesión de identificadores y expresiones (en ese mismo orden) unidos por el operador “:” y separados con comas:

```
Glosario a: b, c: d, e: f
Glosario
      g: h,
      i: j,
      k: l,
      m: n
```

Acceso a miembros con operador de flecha

Para acceder a los miembros de un glosario, se emplea el operador de flecha (“->”).

Si tenemos un Glosario `canasta` con un miembro llamado `manzana` y queremos acceder al mismo, hay que referenciar al Glosario y utilizar el operador de flecha junto al identificador del miembro deseado. Resultando en la expresión `canasta->manzana`.

Intentar acceder un elemento que no existe en un Glosario devolverá `Nada`, y subsecuente uso del operador de flecha alzará un error.

Todos los Glosarios tienen un miembro `->tamaño` indicando su cantidad de miembros actual. Dicha cantidad excluye el miembro `tamaño` en sí.

Valor por Defecto

Un glosario vacío. O sea, sin miembros (y por lo tanto, sin identificadores accesibles).

Ejemplo

```
CARGAR clases con Glosario
    guerrero: (Lista "guerrero", "Loco con un Hacha"),
    arquero: (Lista "arquero", "Tarado que No Sabe Apuntar"),
    mago: (Lista "mago", "← Nunca Tiene Maná"),
    paladín: (Lista "paladín", "Tonto pero No Muere"),
    curandero: (Lista "curandero", "Nadie Quiere Jugar Esto")

CARGAR jugador con Glosario
    nombre: usuario->nombre,
    clase: clases->paladín->0,
    título: clases->paladín->1,
    vida: 150,
    mana: 20,
    aturdido: Falso

COMENTAR "Por ejemplo: 'El jugador Bot de Puré tiene 150 HP'"
ENVIAR "El jugador ", jugador->nombre, " tiene ", jugador->vida, " HP"
```

Conversiones

De glosario sin miembros

- A Texto: `Texto <Glosario vacío>` = `""`.
- A Dupla: `Dupla <Glosario vacío>` = `Falso`.

De glosario con miembros

- A Texto: `Texto (Glosario horas: 3, minutos: 47)` = `"horas: 3, minutos: 47"`.
- A Dupla: `Dupla (Glosario algo: "coso")` = `Verdadero`.

Coma Final Ignorable

Si pones una coma al final de un miembro, justo antes de un nuevo indicador de sentencia o fin de bloque, esta coma será ignorada. Esto es útil cuando programas y se ve mejor:

```
CARGAR glo con Glosario
    a: 1,
    b: 2,
    c: 3,
    d: 4,
    e: 5,

ENVIAR glo->claves() ->unir(", ") COMENTAR "Envía 'a, b, c, d, e'"
ENVIAR glo->valores()->unir(", ") COMENTAR "Envía '1, 2, 3, 4, 5'"
```

Revisa los Métodos de Glosario

Puedes ver todos los métodos de Glosario desde aquí.

Marco

Tipo de dato estructural especial de variable

Representa un marco (Embed) de Discord. Se podría decir que es una variación de los Glosarios que solo acepta unas propiedades particulares, siendo estas las necesarias para enviar un marco.

Uso

Este tipo de dato solo debe ser usado para enviarse en un mensaje con la sentencia ENVIAR. Es una estructura con capacidades de lectura y escritura bastante limitadas por esa y otras cuantas razones.

Las propiedades de un Marco no pueden ser accedidas por medio del operador de flecha, y tampoco pueden ser modificadas con el mismo. Para modificar un marco, debes usar las funciones de Marco que tiene PuréScript.

Expresión Literal

Ninguna. Los marcos solo pueden ser declarados. Puedes modificados por medio de un conjunto de funciones nativas de Marco.

Valor Inicial

Un Marco sin datos asociados.

Propiedades de Marco

Un Marco puede ser asociado a un conjunto de propiedades privadas para modificar la forma en la que se representa en Discord. Esto se logra por medio de la ejecución de funciones nativas de Marco.

Puedes ver todas las propiedades de Marco descritas y asociadas a sus respectivas funciones en la siguiente tabla:

Propiedad	Función Accesora	Descripción
campos	marcoAgregarCampo	Lista de pares nombre-valor como contenido del Marco. Debajo del título/descripción y por encima del pie. Pueden haber hasta 3 por fila.
autor	marcoAsignarAutor	Texto por encima del resto de componentes del Marco. Puede incluir un avatar.
autor→nombre	marcoAsignarAutor	El nombre del autor.
autor→avatar	marcoAsignarAutor	El avatar del autor. A la izquierda del nombre.
color	marcoAsignarColor	Color del Marco. Explicado debajo.
descripción	marcoAsignarDescripción	Descripción justo debajo del título del Marco.
imagen	marcoAsignarImagen	Un enlace a una imagen. Justo por encima del pie y debajo de los campos del Marco.
miniatura	marcoAsignarMiniatura	Un enlace a una imagen. Arriba y a la derecha del resto de componentes del marco.
pie	marcoAsignarPie	Un Texto debajo del resto de componentes del Marco. Puede contener una imagen.
pie→texto	marcoAsignarPie	El Texto del pie de Marco.
pie→imagen	marcoAsignarPie	Una imagen a la izquierda del Texto del pie.
título	marcoAsignarTítulo	Un Texto justo por debajo del autor.

Colores de Marco

El color del Marco define la apariencia de la línea gruesa en el borde izquierdo del mismo. Un Marco puede tener los colores descritos en la tabla de la siguiente página:

Nombre	Código	Muestra / Descripción
colorAleatorio	-	Un color **aleatorio**.
colorAmarillo	"#FFFF00"	
colorAqua	"#1ABC9C"	
colorAquaOscuro	"#11806A"	
colorAzul	"#3498DB"	
colorAzulOscuro	"#206694"	
colorBlanco	"#FFFFFF"	
colorCasiNegro	"#2C2F33"	"*Dark But Not Black*" de **Discord**.
colorDiscord	"#5865F2"	"*Blurple*" de **Discord**.
ColorDorado	"#F1C40F"	
colorDoradoOscuro	"#C27C0E"	
colorFucsia	"#EB459E"	Fucsia oficial de **Discord**.
colorGris	"#95A5A6"	
colorGrisClaro	"#BCC0C0"	
colorGrisNegro	"#7F8C8D"	
colorGrisOscuro	"#979C9F"	
colorGríspura	"#99AAB5"	"*Greyple*" de **Discord**.
colorMarino	"#34495E"	
colorMarinoOscuro	"#2C3E50"	
colorNaranja	"#E67E22"	
colorNaranjaOscuro	"#A84300"	
colorNegro	"#23272A"	"*Not Quite Black*" de **Discord**.
colorPúrpura	"#9B59B6"	
colorPúrpuraOscuro	"#71368A"	
colorRojo	"#ED4245"	Rojo oficial de **Discord**.
colorRojoOscuro	"#992D22"	
colorRosaClaro	"#E91E63"	
colorRosaOscuro	"#AD1457"	
colorVerde	"#57F287"	Verde oficial de **Discord**.
colorVerdeOscuro	"#1F8B4C"	

Entrada

Tipo de dato variable de origen externo

Representa un valor primitivo ingresado por el Usuario. Solo puede especificarse en la sentencia REGISTRAR. El tipo que representa será decidido por el valor primitivo asignado durante la primera ejecución.

Expresión de Entrada

Adopta la expresión ingresada por el Usuario.

Valor por defecto

En la primera ejecución, una Entrada adopta el valor especificado por el Usuario en la sentencia REGISTRAR, conocido como el "valor por defecto". En ejecuciones subsecuentes está garantizado que su valor será, al menos, del tipo del valor de prueba.

Ejemplos

- `42`
- `"Hola"`
- `Verdadero`

Conversiones

Una Entrada adopta las conversiones del tipo de valor primitivo asociado a esta.

Función

Tipo de procedimiento invocable variable

Representa una sucesión de sentencias a ejecutar cuando se llama/invoca a la Función a través de un identificador. Existen una multitud de Funciones nativas en PuréScript.

Expresión literal

Para expresar una Función, se usa el indicador de tipo de Función seguido del identificador de la Función y sus argumentos entre paréntesis (en ese mismo orden). El cuerpo de la Función descrita se delimita con los indicadores de sentencia BLOQUE y FIN. El cuerpo de la Función contiene todas las sentencias a ejecutar cuando se la invoque en otra parte del Programa.

```
Función fn(argumentos...) BLOQUE
    ...sentencias
FIN
```

Nótese que esta sintaxis es solo la de **expresión** de Función, y difiere de la sintaxis de **registro** de Función que se ve en la sentencia REGISTRAR.

Valor por defecto

`Función <Identificador>() BLOQUE FIN`.

Ejemplos de Argumentos y Cuerpo de Función

- `() BLOQUE ENVIAR "¡Hola!" FIN`
- `(a, b) BLOQUE DEVOLVER "El resultado es: ", a + b FIN`
- `(li) BLOQUE SUMAR li->el FIN`
- `(a) BLOQUE CARGAR b con a CARGAR c con a * b FIN`

Conversiones

Una Función no puede convertirse a ningún otro tipo.

Funciones y Métodos

Las funciones y métodos en PuréScript son básicamente lo mismo, y se declaran de la misma manera. La diferencia es que con "método" nos referimos a una función que pertenece o está asociada a un contenedor que puede ser primitivo en caso de métodos de tipo o una estructura. Un ejemplo recurrente de esto sería el método de Lista: `Lista->unir(separador)`, que une todas las representaciones de Texto de sus elementos en un solo Texto con el `separador` especificado.

Nótese que este ejemplo trata de un método de Lista nativo. Puedes definir tus propios métodos de igual forma, usando el operador de flecha para asignar una Función a un miembro.

Argumentos por defecto

Cada identificador de argumento puede llevar un valor por defecto en caso de que no se pase el valor requerido.

Para obtener este comportamiento, simplemente coloca dos puntos y luego el valor.

Nada

Valor primitivo especial de variable

El valor por defecto de Números y Textos que no han sido inicializados. También es el valor que se asume de los elementos de Lista o miembros de Glosarios que no han sido declarados.

Expresión literal

```
Nada
```

Conversiones

- A Texto: `Texto Nada` = `"Nada"`.
- A Dupla: `Dupla Nada` = `Falso`.

Ejemplo

El siguiente código enviará “Nada” en el chat, ya que `x` no está inicializado y su valor por defecto `Nada` se convierte a “nada”.

```
CREAR Número x
ENVIAR Texto x
```

Comprobando Nada en una Variable

Puedes usar la función `esNada(x)` para comprobar si una variable, un elemento o un miembro vale `Nada`.

Estructuras Estandarizadas

Listado categorizado ordenado

Los siguientes son ejemplos de estructuras comunes de conseguir al trabajar con ciertas funciones o mencionar algunas variables nativas:

Glosario "Miembro"

Un Glosario 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.

Glosario "Canal"

Un Glosario 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.

Glosario "Rol"

Un Glosario 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.

Glosario "Servidor"

Un Glosario 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`	`Glosario /Nada`	Un Glosario 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.
`imagenInvitación`	`Texto`	El enlace de la imagen de invitación del servidor.
`dueño`	`Glosario`	Un Glosario representando el miembro dueño del servidor.

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 "->". Revisa la tabla:

Propiedad	Tipo	Descripción
pi	Número	La relación entre la circunferencia de un círculo y su diámetro. Equivalente aproximadamente a `3.141592`.
usuario	Glosario	El miembro que llama al Tubérculo.
canal	Glosario	El canal en el que se llama al Tubérculo.
servidor	Glosario	El servidor al que pertenece el Tubérculo.
color(...)	Texto	Puedes ver la lista completa de variables de color aquí.
Número	Glosario de tipo	El contenedor de los métodos accesibles por medio de cualquier Número.
Texto	Glosario de tipo	El contenedor de los métodos accesibles por medio de cualquier Texto.
Lista	Glosario de tipo	El contenedor de los métodos accesibles por medio de cualquier Lista.
Glosario	Glosario de tipo	El contenedor de los métodos accesibles por medio de cualquier Glosario.

Funciones Nativas

Listado categorizado ordenado

Las siguientes funciones nativas son accesibles de forma global. Estas no se acceden con el operador de flecha "->".

Con tal de dejar claro el uso de las funciones, se pondrá "*" siguiendo los argumentos obligatorios y se indicará el tipo o los tipos de valor admitibles de cada argumento luego de ":". Seguido de la expresión de una función, se especificará el tipo o los tipos de valor que puede devolver.

Nótese que tú no debes poner "*" ni ":" ni los tipos que se ven a continuación. Estos son meramente ilustrativos y con el fin de informar el uso de dichas funciones.

Utilidad General

`dado(a: Número, b: Número/Dupla, haciaArriba: Dupla) → Número`

Argumento	Tipo	Por defecto	Descripción
a	Número	1	El inicio del rango aleatorio, o el final si no se ingresa `b`.
b	Número /Dupla		El final del rango aleatorio, o `haciaArriba`.
haciaArriba	Dupla		Si redondear hacia arriba o hacia abajo.

Tira un dado imaginario y devuelve un Número aleatorio.

Si no se ingresa `a`, el dado varía entre 0 inclusive y 1 exclusive.

Si no se ingresa `b`, el dado varía entre 0 inclusive y `a` exclusive.

Si `b` es un Número, el dado tira entre `a` inclusive y `b` exclusive.

Si `b` o `haciaArriba` son una `Dupla`, el valor del dado se redondeará hacia arriba si dicha Dupla es `Verdadero` o hacia abajo si dicha Dupla es `Falso`.

Devuelve – Un Número aleatorio que cumple con los parámetros ingresados.

Comprobación de Tipo

Estas funciones aceptan cualquier valor y devuelven si es del tipo cuestionado o no.

`esNúmero(x*) → Dupla`

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*) → Dupla`

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.

`esDupla(x*) → Dupla`

Argumento	Tipo	Por defecto	Descripción
x	Cualquier	Requerido	El valor a comprobar.

Verifica si `x` es de tipo Dupla.

Devuelve – `Verdadero` si `x` es una Dupla; `Falso` en cualquier otro caso.

`esLista(x*) → Dupla`

Argumento	Tipo	Por defecto	Descripción
x	Cualquier	Necesario	El valor a comprobar.

Verifica si `x` es de tipo Lista.

Devuelve – `Verdadero` si `x` es una Lista; `Falso` en cualquier otro caso.

`esGlosario(x*) → Dupla`

Argumento	Tipo	Por defecto	Descripción
x	Cualquier	Requerido	El valor a comprobar.

Verifica si `x` es de tipo Glosario.

Devuelve – `Verdadero` si `x` es un Glosario; `Falso` en cualquier otro caso.

`esMarco(x*) → Dupla`

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*) → Dupla`

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.

Modificación de Marcos

Estas funciones modifican el Marco ingresado en lugar de devolver un valor.

`marcoAgregarCampo(marco*: Marco, nombre*: Texto, valor*: Texto, alineado: Dupla) → Nada`

Argumento	Tipo	Por defecto	Descripción
marco	Marco	Requerido	El Marco a modificar.
nombre	Texto	Requerido	El nombre del campo a añadir.
valor	Texto	Requerido	El valor del campo a añadir.
alineado	Dupla	Falso	Si va en la misma columna que otros campos.

Añade un campo al `marco` con el `nombre` y el `valor` especificados. No pueden ser Textos vacíos. Pueden haber de 2 a 3 campos `alineados` en la misma columna.

`marcoAsignarAutor(marco*: Marco, nombre*: Texto, ícono: Texto) → Nada`

Argumento	Tipo	Por defecto	Descripción
marco	Marco	Requerido	El Marco a modificar.
nombre	Texto	Requerido	El nombre del autor a asignar.
ícono	Texto		El ícono del autor a asignar.

Asigna un autor al `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.

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).

`marcoAsignarColor(marco*: Marco, color*: Texto) → Nada`

Argumento	Tipo	Por defecto	Descripción
`marco`	`Marco`	**Requerido**	El Marco a modificar.
`color`	`Texto`	**Requerido**	El color a asignar.

Asigna el `color` del `marco`. No se puede pasar un Texto vacío como color. Puedes ingresar un código hexadecimal o un color predeterminado.

Por defecto, el color de un Marco es el código hexadecimal `"#000000"` (negro puro). Usar esta Función reemplazará el color actual.

`marcoAsignarDescripción(marco*: Marco, descripción*: Texto) → Nada`

Argumento	Tipo	Por defecto	Descripción
`marco`	`Marco`	**Requerido**	El Marco a modificar.
`descripción`	`Texto`	**Requerido**	La descripción a asignar.

Asigna una `descripción` al `marco`. No se puede pasar una descripción vacía. Si el Marco ya tenía una descripción, se la reemplaza.

`marcoAsignarImagen(marco*: Marco, imagen*: Texto) → Nada`

Argumento	Tipo	Por defecto	Descripción
`marco`	`Marco`	**Requerido**	El Marco a modificar.
`imagen`	`Texto`	**Requerido**	El enlace de imagen a asignar.

Asigna una `imagen` al `marco`. El Texto de la imagen debe ser un enlace válido. Si el Marco ya tenía una imagen, se la reemplaza.

`marcoAsignarMiniatura(marco*: Marco, miniatura*: Texto) → Nada`

Argumento	Tipo	Por defecto	Descripción
`marco`	`Marco`	**Requerido**	El Marco a modificar.
`miniatura`	`Texto`	**Requerido**	El enlace de imagen de miniatura a asignar.

Asigna una imagen de `miniatura` al `marco`. El Texto de la miniatura debe ser un enlace válido. Si el Marco ya tenía una miniatura, se la reemplaza.

`marcoAsignarPie(marco*: Marco, texto*: Texto, ícono: Texto) → Nada`

Argumento	Tipo	Por defecto	Descripción
marco	Marco	Requerido	El Marco a modificar.
texto	Texto	Requerido	El texto del pie a asignar.
ícono	Texto		El enlace del ícono del pie a asignar.

Asigna un pie al `marco` con el `texto` y el `ícono` especificados. No se puede pasar un texto vacío y el enlace de la imagen debe además ser válido. Si el Marco ya tenía un pie, se lo reemplaza.

`marcoAsignarTítulo(marco*: Marco, título*: Texto) → Nada`

Argumento	Tipo	Por defecto	Descripción
marco	Marco	Requerido	El Marco a modificar.
título	Texto	Requerido	El título a asignar.

Asigna un `título` al `marco`. No se puede pasar un título vacío. Si el Marco ya tenía un título, se lo reemplaza.

Utilidad de Discord

`buscarMiembro(búsqueda*: Texto) → Glosario o 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 caracter está el fragmento de Texto buscado.

Devuelve – Un Glosario del miembro encontrado, o `Nada` si no se encuentra.

`buscarCanal(búsqueda*: Texto) → Glosario o 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 Glosario del canal encontrado, o `Nada` si no se encuentra.

`buscarRol(búsqueda*: Texto) → Glosario o 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 Glosario del rol encontrado, o `Nada` si no se encuentra.

Otras Comprobaciones

`esEnlace(x*) → Dupla`

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.

Devuelve – `Verdadero` si `x` es un Texto de enlace; `Falso` en cualquier otro caso.

`esArchivo(x*) → Dupla`

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*) → Dupla`

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

Los siguientes métodos nativos son accesibles desde variables, estructuras y valores varios.

Si bien los miembros suelen estar relacionados a Glosarios, la mayoría de valores heredan métodos miembros específicos a su tipo de valor. Estos, al ser miembros, se acceden por medio del operador de flecha "->".

Cabe mencionar que los métodos miembros listados a continuación no se consideran miembros genuinos. Por ejemplo: incluso si tienes acceso a 3 miembros métodos en un Glosario, el Glosario va a seguir teniendo tamaño 0 si no se le carga ningún miembro.

Para demostrar de forma comprensiva cada método, se escribirá la expresión de flecha completa a modo de ejemplo. Se pondrá "*" siguiendo los argumentos obligatorios y se indicará el tipo o los tipos de valor admitibles de cada argumento luego de ":". Seguido de la expresión de un método, se especificará el tipo o los tipos de valor que puede devolver.

Nótese que tú no debes poner "*" ni ":" ni los tipos que se ven a continuación. Estos son meramente ilustrativos y con el fin de informar del uso de los métodos.

Métodos de Número

Los siguientes métodos pueden ser accedidos desde cualquier Número:

```
unNúmero->redondear(haciaArriba: Dupla) → Número
```

Argumento	Tipo	Por defecto	Descripción
`haciaArriba`	`Dupla`		Si redondear hacia arriba o hacia abajo.

> Redondea `unNúmero` al entero más cercano. Si se pasa una Dupla en `haciaArriba`, se redondea hacia arriba si es `Verdadero` o hacia abajo si es `Falso`.
>
> **Devuelve** – `unNúmero` redondeado como se especifica.

```
unNúmero->aTexto(n: Número) → Texto
```

Argumento	Tipo	Por defecto	Descripción
n	Número		La precisión del Número.

Crea una representación de Texto de `unNúmero` con `n` dígitos significativos.

Si no se pasa `n`, funciona igual que la conversión `(Texto unNúmero)`.

Devuelve – `unNúmero` con `n` cifras significativas. En forma de Texto.

Métodos de Texto

Los siguientes métodos pueden ser accedidos desde cualquier Texto:

```
unTexto->caracterEn(posición*: Número) → Texto/Nada
```

Argumento	Tipo	Por defecto	Descripción
posición	Número	Requerido	La posición del caracter a encontrar.

Crea un nuevo Texto a partir de un solo caracter de `unTexto`. El caracter seleccionado se decide por la `posición` especificada. Estando el primer caracter en la posición `0`, el segundo en la posición `1`, el tercero en `2` y así.

Devuelve – El caracter de `unTexto` en el índice especificado, o `Nada` si no existe.

```
unTexto->posiciónDe(caracter*: Texto) → Número
```

Argumento	Tipo	Por defecto	Descripción
caracter	Texto	Requerido	El caracter a encontrar.

Crea un nuevo Número dictando la posición en la que se encuentra el `caracter` buscado en `unTexto`. Se devuelve la primer posición en la que se encuentra el caracter, leyendo de izquierda a derecha.

Devuelve – La posición del `caracter` en `unTexto`, o `-1` si no existe.

`unTexto->últimaPosiciónDe(caracter*: Texto) → Número`

Argumento	Tipo	Por defecto	Descripción
caracter	Texto	Requerido	El caracter a encontrar.

Mismo comportamiento que `unTexto->posiciónDe(caracter)`, pero se devuelve la primer posición encontrada leyendo de derecha a izquierda.

Devuelve – La última posición del `caracter` en `unTexto`, o `-1` si no existe.

`unTexto->comienzaCon(subTexto*: Texto) → Dupla`

Argumento	Tipo	Por defecto	Descripción
subTexto	Texto	Requerido	El sub-texto a comprobar.

Analiza si `unTexto` comienza inmediatamente con el `subTexto` especificado.

Devuelve – `Verdadero` si `unTexto` comienza con el sub-texto especificado.

`unTexto->terminaCon(subTexto*: Texto) → Dupla`

Argumento	Tipo	Por defecto	Descripción
subTexto	Texto	Requerido	El sub-texto a comprobar.

Analiza si `unTexto` termina inmediatamente con el `subTexto` especificado.

Devuelve – `Verdadero` si `unTexto` termina con el sub-texto especificado.

`unTexto->incluye(subTexto*: Texto) → Dupla`

Argumento	Tipo	Por defecto	Descripción
subTexto	Texto	Requerido	El sub-texto a comprobar.

Analiza si `unTexto` contiene en algún lado el `subTexto` especificado.

Devuelve – `Verdadero` si `unTexto` incluye el sub-texto especificado.

```
unTexto->repetir(veces*: Número) → Texto
```

Argumento	Tipo	Por defecto	Descripción
veces	Número	0	La cantidad de veces a repetir el Texto.

Crea un Texto basado en repetir `unTexto` la especificada cantidad de `veces`. El Número ingresado debe ser `0` ó más, y no debería ser muy grande.

Si el Número es `0`, se crea un Texto vacío. Si el Número es `1`, se crea un Texto idéntico a `unTexto`.

Devuelve – `unTexto` repetido la especificada cantidad de veces.

```
unTexto->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 las `ocurrencias` encontradas en `unTexto` por el `reemplazo` especificado.

Devuelve – `unTexto` con las ocurrencias sustituidas por el valor de reemplazo.

```
unTexto->partir(separador*: Texto) → Lista de Textos
```

Argumento	Tipo	Por defecto	Descripción
separador	Texto	Requerido	El Texto a buscar para partir en elementos.

Crea una Lista con los contenidos de `unTexto` separados en elementos por cada ocurrencia del `separador` especificado. Las ocurrencias del `separador` como tal no son agregadas a la Lista.

Devuelve – Una lista con los elementos de `unTexto` separados acordemente.

`unTexto->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 recorte. Exclusive.

Crea un Texto a partir de un recorte de `unTexto` de `inicio` a `fin`, sin incluir el caracter en `fin`. La posición se cuenta desde `0` para el primer caracter.

Si el Texto está vacío, se crea un Texto vacío.

Si `inicio` excede el largo del Texto, se crea un Texto vacío.

Si no se pasa `fin`, o este excede el largo del Texto, no se recortará el final.

Si alguna de 2 posiciones es un Número negativo, esta se contará desde el último caracter hacia atrás.

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 – `unTexto` recortado de `inicio` a `fin` apropiadamente.

`unTexto->aMinúsculas() → Texto`

Crea una versión de `unTexto` en la cual todos los caracteres en mayúsculas se pasan a minúsculas. Obviamente no afecta caracteres no-alfabéticos.

Devuelve – `unTexto` en minúsculas.

`unTexto->aMayúsculas() → Texto`

Bueno... te puedes imaginar lo que hace, ¿no?

Devuelve – Exactamente eso.

`unTexto->normalizar() → Texto`

Crea una versión de `unTexto` sin caracteres en blanco al inicio ni al final. Los caracteres en blanco entre medio de otros caracteres no son eliminados.

Devuelve – `unTexto` sin caracteres en blanco envolventes.

`unTexto->aLista() → Lista de Textos`

Crea una Lista a partir de los caracteres de `unTexto`. El largo de la Lista equivaldrá al largo del Texto.

Devuelve – `unTexto` en forma de Lista de caracteres.

Métodos de Lista

Los siguientes métodos pueden ser accedidos desde cualquier Lista:

`unaLista->unir(separador: Texto) → Texto`

Argumento	Tipo	Por defecto	Descripción
separador	Texto	","	El Texto a colocar entre los elementos unidos.

Crea un Texto con todos los elementos de `unaLista` representados en Texto y separados por el `separador` especificado.

Si la Lista está vacía, se crea un Texto vacío.

Puedes pasar `""` para imitar el comportamiento de `(Texto unaLista)`.

Devuelve – `unaLista` con todos sus elementos unidos en forma de Texto.

`unaLista->vacía() → Dupla`

Analiza si la lista no tiene elementos.

Devuelve – `Verdadero` si `unaLista` está vacía.

`unaLista->incluye(valor*) → Dupla`

Argumento	Tipo	Por defecto	Descripción
valor	Cualquier	Requerido	El valor a comprobar entre los elementos.

Analiza si `unaLista` contiene un elemento con el `valor` especificado.

Devuelve – `Verdadero` si `unaLista` contiene el valor especificado.

`unaLista->invertir() → Lista`

Crea una Lista a partir de `unaLista` con todos los elementos ordenados al revés.

Devuelve – `unaLista` al revés.

`unaLista->ordenar() → Lista`

Crea una Lista a partir de `unaLista` con todos los elementos convertidos a Texto y reordenados en base al orden Unicode de sus valores.

Devuelve – `unaLista` ordenada "alfabéticamente".

`unaLista->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.

Crea una Lista a partir de un recorte de `unaLista` de `inicio` a `fin`, sin incluir el elemento en la posición `fin`.

Si la Lista está vacía, se crea una Lista vacía.

Si `inicio` excede el largo de la Lista, se crea una Lista vacía.

Si no se pasa `fin`, o este excede el largo de la Lista, no se recortará 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 – `unaLista` recortada de `inicio` a `fin` apropiadamente.

`unaLista->último() → Cualquier`

Crea un valor correspondiente al último elemento de `unaLista`.

Devuelve – El último valor de `unaLista`.

`unaLista->robarPrimero() → Cualquier`

Remueve el primer elemento de `unaLista` y crea un valor con este.

Devuelve – El elemento eliminado de `unaLista`.

`unaLista->robarÚltimo() → Cualquier`

Remueve el último elemento de `unaLista` y crea un valor con este.

Devuelve – El elemento eliminado de `unaLista`.

`unaLista->robar(posición*: Número) → Cualquier`

Argumento	Tipo	Por defecto	Descripción
posición	Número	Requerido	El valor a comprobar entre los elementos.

Remueve el elemento en la `posición` especificada de `unaLista` y crea un valor con este.

Devuelve – El elemento eliminado de `unaLista`.

`unaLista->aGlosario() → Glosario`

Crea un Glosario a partir de los elementos de `unTexto`. El tamaño del Glosario equivaldrá al largo de la Lista.

Los nombres de los miembros del glosario corresponderán al índice de cada elemento con una "v" al inicio, y almacenarán los valores de esos respectivos índices.

Devuelve – `unaLista` en forma de Glosario.

Métodos de Glosario

Los siguientes métodos pueden ser accedidos desde cualquier Glosario:

`unGlosario->nombres() → Lista de Textos`

> Crea una Lista con los identificadores de todos los miembros de `unGlosario`.

Devuelve – Una Lista con los nombres de los miembros de `unGlosario`.

`unGlosario->valores() → Lista`

> Crea una Lista con los valores de todos los miembros de `unGlosario`.

Devuelve – Una Lista con los valores de los miembros de `unGlosario`.

`unGlosario->miembros() → Lista de Listas`

> Crea una Lista con Listas que contienen el nombre y el valor de todos los miembros de `unGlosario`.
>
> La Lista que contiene otras Listas tiene un largo equivalente al tamaño del Glosario.
>
> Cada Lista interna tiene un largo de `2` y representa los pares nombre/valor de uno de los miembros del Glosario. La posición `0` contiene el nombre y la posición `1` contiene el valor.

Devuelve – Una Lista con los pares nombre/valor de los miembros de `unGlosario`.

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 de la operación.

Se hará referencia al resultado de la operación como la "evaluación" de esta.

Los operadores binarios siguen la sintaxis `<izquierda> <operador> <derecha>`, mientras que los operadores unarios siguen la sintaxis `<operador> <argumento>`.

Operador de Asignación "con"

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	con	Variable	Cualquier	N/A

Asigna el valor del operando derecho al izquierdo.

Operadores Aritméticos

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	Convertible a Número	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 potencia de la potenciación.

Tipo	Operador	Argumento	Evaluación
Unario	`+`	Convertible a Número	Número

Evalúa al Número actual del operando derecho. Es una forma rápida de representar valores como Números.

Tipo	Operador	Argumento	Evaluación
Unario	`-`	Convertible a Número	Número

Evalúa al Número opuesto del operando derecho.

Operador de Concatenación

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	`:`	`Variable`	`Cualquier`	`N/A`

Se usa en expresiones literales de Glosario para asignar miembros a estos. El operando izquierdo representa el identificador del miembro, mientras que el operando derecho representa el valor que le corresponde a dicho miembro.

Operador de Flecha "->"

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	`->`	`Cualquier`	`Cualquier`	`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 del miembro, mientras que el operando derecho representa el miembro-propiedad accedido.

La operación evalúa al valor del miembro-propiedad dentro del contenedor referenciado.

Se pueden concatenar múltiples operaciones de flecha para acceder progresivamente a los miembros dentro de los miembros del contenedor principal.

Operadores Lógicos

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	`y`	`Convertible a Dupla`	`Convertible a Dupla`	`Cualquier`

En términos simples, evalúa si ambos operandos evalúan a `Verdadero`. Incluye evaluación de cortocircuito.

Formalmente:

- 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	`o`	`Convertible a Dupla`	`Convertible a Dupla`	`Cualquier`

En términos simples, evalúa si al menos uno de los operandos evalúa a `Verdadero`. Incluye evaluación de cortocircuito.

Formalmente:

- Si el operando izquierdo evalúa a `Falso`, se evalúa al operando izquierdo.
- Si el operando izquierdo evalúa a `Verdadero`, se evalúa al operando derecho.

Tipo	Operador	Argumento	Evaluación
Unario	`no`	`Convertible a Dupla`	`Dupla`

Invierte el valor de Dupla del argumento. Evalúa al resultado de la inversión.

- Si el argumento evalúa a `Verdadero`, la operación evalúa a `Falso`.
- Si el argumento evalúa a `Falso`, la operación evalúa a `Verdadero`.

Operadores Comparativos

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	`es`	`Cualquier`	`Cualquier`	`Dupla`

Si ambos operandos evalúan al mismo tipo y valor, se evalúa a `Verdadero`. Si no coinciden en tipo, valor, o ambos, se evalúa a `Falso`.

Nótese que con "mismo valor" en este caso se hace referencia a valores intercambiables indiferentemente del tipo del valor. Por ejemplo, un Texto `"42"` puede convertirse al Número `42`, así como una Dupla `Falso` puede convertirse al Número `0`.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	`no es`	`Cualquier`	`Cualquier`	`Dupla`

Considera el mismo criterio que `es` e invierte el resultado.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	`parece`	`Cualquier`	`Cualquier`	`Dupla`

Si ambos operandos evalúan al mismo valor, se evalúa a `Verdadero`. Si no coinciden en valor, se evalúa a `Falso`. Si coinciden en valor pero no en tipo, se evalúa a `Verdadero`.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	`no parece`	`Cualquier`	`Cualquier`	`Dupla`

Considera el mismo criterio que `parece` e invierte el resultado.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	`precede`	`Cualquier`	`Cualquier`	`Dupla`

Si el operando izquierdo tiene un "menor valor" que el derecho, evalúa a `Verdadero`. Alternativamente, evalúa a `Falso`.

Este "menor valor" puede significar un menor Número, un Texto que precede alfabéticamente a otro, `Falso` antes que `Verdadero`, etc.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	`no precede`	`Cualquier`	`Cualquier`	`Dupla`

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`	`Dupla`

Si el operando izquierdo tiene un "mayor valor" que el derecho, evalúa a `Verdadero`. Alternativamente, evalúa a `Falso`.

Este "mayor valor" puede significar un mayor Número, un Texto que sucede alfabéticamente a otro, `Verdadero` luego de `Falso`, etc.

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	`no excede`	`Cualquier`	`Cualquier`	`Dupla`

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.

Operadores de Agrupación "()"

Tipo	Operador	Izquierda	Derecha	Evaluación
Encapsulación	`(`		`Cualquier`	`Cualquier`
Encapsulación	`)`	`Cualquier`		`Cualquier`

Encapsula expresiones y operaciones y les da menor precedencia sobre toda operación.

Dichas expresiones y operaciones serían el operando que va entre los paréntesis.

Operador de Coma ","

Tipo	Operador	Izquierda	Derecha	Evaluación
Binario	`,`	`Cualquier`	`Cualquier`	`Cualquier`

Se usa en literales de Lista, literales de Glosario, expresiones de argumentos de Función y plantillas de Texto como un medio para separar expresiones en un formato de listado.

Dichas expresiones serían los operandos que rodean la coma.

Tabla de Precedencias

Listado categorizado ordenado

Las operaciones 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. Así mismo, también se define si las evaluaciones en una misma precedencia se evalúan de izquierda a derecha o de derecha a izquierda.

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, como el orden de precedencia indica que primero se debe realizar la multiplicación: si no colocamos paréntesis, esto evalúa a `14`.

Así mismo, cuando las precedencias coinciden, se recurre a la asociatividad.

La asociatividad a la izquierda significa que las operaciones más a la izquierda se evalúan primero, mientras que la asociatividad a la derecha significa que las operaciones más a la derecha se evalúan primero.

Múltiples sumas y restas se evalúan de izquierda a derecha por tener la misma precedencia (o sea, tienen asociatividad a la izquierda). Pasaría los mismo con múltiples multiplicaciones y divisiones. Por ejemplo:

```
2 + 4 - 1 + 3 = 8
3 * 4 / 2 * 5 = 30
```

Podemos ilustrar el orden de evaluación con paréntesis de la siguiente forma:

```
((2 + 4) - 1) + 3 = 8
((3 * 4) / 2) * 5 = 30
```

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:

```
(5 + ((4 * (3 ^ 2)) / 2)) - 1 = 22
```

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
11	A la derecha (!)[3]	`… -> …`	Acceso a Miembros
	A la izquierda	`… ( … )`	Llamado a Función
10	A la derecha	`no …`	NO lógico
		`+ …`	Suma Unaria
		`- …`	Resta Unaria
9	A la derecha	`… ^ …`	Potenciación
8	A la izquierda	`… * …`	Multiplicación
		`… / …`	División
		`… % …`	Módulo
7	A la izquierda	`… + …`	Adición/Concatenación
		`… - …`	Sustracción
6	A la izquierda	`… precede …`	Comparación
		`… no precede …`	
		`… excede …`	
		`… no excede …`	
5	A la izquierda	`… es …`	Igualdad
		`… no es …`	Desigualdad
		`… parece …`	Similaridad
		`… no parece …`	De-similaridad
4	A la izquierda	`… y …`	Y lógico
3	A la izquierda	`… o …`	O lógico
2	N/A	`… con …`	Asignación
1	A la izquierda	`… , …`	Coma

3 Esta asociatividad cambiará a la izquierda en una futura versión de PuréScript, con tal de no tener que usar paréntesis para acceder métodos-miembros de miembros de estructuras y otros inconvenientes.

Definiciones

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

Ilustración https://twitter.com/pu_yan8901

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.

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 nuestro programa.

La única ocasión en la que puedes expresar un literal de Texto (o un Número) como un identificador, es cuando usas el operador de flecha para expresar un acceso de miembro: `"estructura"->dato`, `estructura->"dato"`, `estructura->0` o `0->dato.`

Primitivo/a

Un valor primitivo es un dato que no es una estructura y no tiene métodos ni propiedades como se las definiría en una Lista o Glosario.

PuréScript cuenta con 3 primitivas:

- Número
- Texto
- Dupla

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 una primitiva. La variable puede ser cargada con un nuevo valor, pero el valor existente no puede ser manipulado en formas que se lo vería con Listas o Glosarios.

Nótese cómo, por ejemplo, métodos como `Número->aTexto(n)` devuelven un nuevo Texto en lugar de transformar el Número en un Texto, devolver ese valor y devolver el Número a su estado original.

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
- Glosario
- 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`.