



Introducción a la Programación

Clases teóricas

por Pablo E. “Fidel” Martínez López

10. Listas





Repaso



- **Programar es comunicar** (con máquinas y personas)
 - Estrategia de solución (división en subtarear)
 - Legibilidad (elección de nombres, indentación)
 - **CONTRATOS:** Propósito, parámetros y precondiciones (y resultado en funciones)
- **Programas** (texto con diversos elementos)
 - **Comandos:** describen acciones
 - **Expresiones:** describen información
 - **Tipos:** clasifican expresiones



- **Comandos**

- Primitivos y secuencia
- PROCEDIMIENTOS (con y sin parámetros)
- Repetición simple (**repeat**)
- Alternativa condicional (**if-then-else**)
- Repetición condicional (**while**)
- Asignación de variables (**:=**)



● Expresiones

- Valores literales y expresiones primitivas
- Operadores
numéricos, de enumeración, de comparación, lógicos
- Alternativa condicional en expresiones (**choose**)
- FUNCIONES (con y sin parámetros, con y sin procesamiento)
- Parámetros y variables (como datos)
- **Constructores (de registros y variantes)**
- **Funciones observadoras de campo**



- **Tipos de datos**

- **Básicos**

- Colores, Direcciones, Números, Booleanos

- **Compuestos**

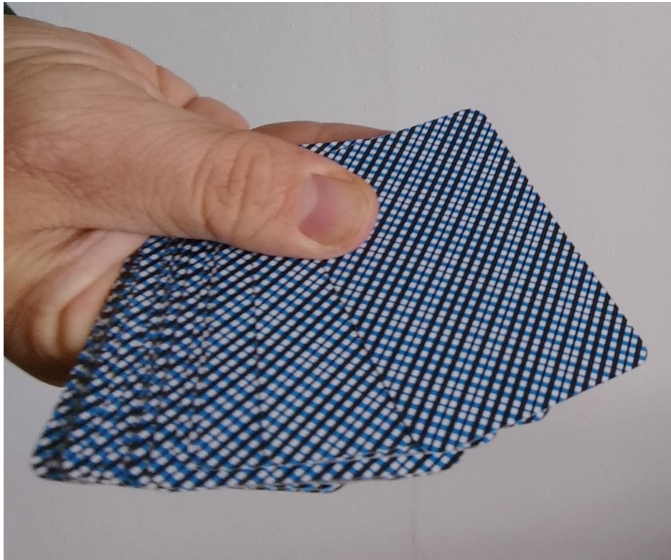
- **Registros** (muchas partes de diferentes tipos)

- **Variantes** (una parte, muchas posibilidades)



Listas

- Si queremos programar un juego de cartas, ¿cómo representar un mazo de cartas?
 - No es un color, ni un número, ni otro tipo básico
 - No es un variante, porque tiene varios elementos (las cartas)
 - No es un registro, porque sus elementos son del mismo tipo
 - Necesitamos un nuevo **tipo de datos**



¡No siempre hay la misma cantidad de elementos!



- Las **listas** son datos con estructura
 - Tienen muchos elementos, todos del mismo tipo
 - Son datos, por lo que se pueden pasar como argumentos, guardar en variables, retornar como resultados, etc.
 - Hay funciones para manipular listas
 - ¿Qué necesitamos para escribir listas?

Estas son funciones primitivas...
No podemos hacerlas nosotros
con las herramientas actuales

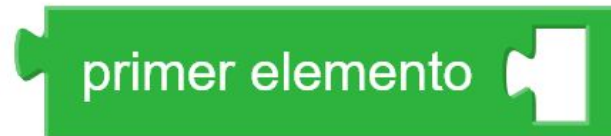
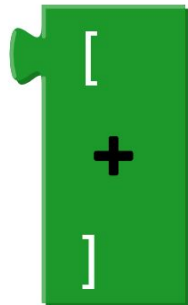
mazo actual

cantidad de elementos

lista de números del escenario



- Precisamos **sintaxis** para listas
 - ¿Cómo decimos cuál es la lista?
 - **Constructores** de listas
 - ¿Cómo obtenemos información de ella?
 - **Funciones de acceso** (primitivas)



Es mejor trabajar en texto



- Para construir listas de forma literal enumeramos sus elementos entre corchetes
 - `[<exp1> , <exp2> , ... , <expN>]`
 - Las expresiones deben ser todas del mismo tipo
 - ¡Puede haber listas con un elemento e incluso con cero!

```
[ manchú(), serena() ]
```


```
[ 10, 20, 30, 40, 50, 60, 70 ]
```

```
[ Norte, Norte, Este, Este, Sur, Sur, Este, Este ] []
```

```
[ Carta(palo <- Espadas, número <- 1)  
  , Carta(palo <- Espadas, número <- 7)  
  , Carta(palo <- Oros,      número <- 7)  
]
```

```
[ 5 ]
```

```
[ 42, 666, 99, 17 ]
```



Esta es una lista
SIN ELEMENTOS,
la *lista vacía*

- El tipo de una lista es “*Lista de*” el tipo de sus elementos
 - `[42 , 99]` es de tipo *Lista de Números* o `[Número]`
 - `[Norte]` es de tipo *Lista de Direcciones* o `[Dirección]`
 - `[Carta (palo<-Espadas , número<-1)]` es de tipo *Lista de Cartas* o `[Carta]`

`[5]`

es una *Lista de Números*
(con un solo número)

`5`

es un *Número*

¡No es lo mismo un
elemento que una
lista con un solo
elemento!



- Otra forma de construir listas es juntando dos existentes
- Para eso se usa el **operador para agregar (*append*)**
 - $\langle expLista1 \rangle ++ \langle expLista2 \rangle$
 - El resultado es una lista que tiene los elementos de ambas, en el mismo orden
 - Es una operación asociativa

```
segundaMitad(mazo) ++ primeraMitad(mazo)
```

```
[ 1, 2, 3, 4 ] ++ [ 5, 6, 7 ]
```

```
cartasNuevas ++ cartasViejas
```

```
[ 0 ] ++ listaAnterior
```

Tiene más sentido
cuando una de las listas
no es literal

- Combinando listas literales y *append* se pueden construir otras operaciones interesantes, por ejemplo
 - `con_agregadoAdelanteDeLaLista_`
que dados un `elemento` y una `lista`, lo agrega adelante
 - Lo abreviamos `cons(elemento, lista)`



```
cons(10, [20, 30, 40])
```

es equivalente a

```
[10, 20, 30, 40]
```

```
cons(Norte, [Este, Sur])
```

es equivalente a

```
[Norte, Este, Sur]
```



- Combinando listas literales y *append* se pueden construir otras operaciones interesantes, por ejemplo
 - `con_agregadoAdelanteDeLaLista_`
que dados un `elemento` y una `lista`, lo agrega adelante
 - Lo abreviamos `cons(elemento, lista)`

```
function cons(elemento, lista) {  
  /* PROPÓSITO: Describir la lista resultante de agregar el  
  elemento dado al principio de la lista dada.  
  PRECONDICIONES: Ninguna.  
  PARÁMETROS:  
  * elemento: Un tipo cualquiera.  
  * lista: Una lista de elementos  
    del mismo tipo que **elemento**.  
  TIPO: El mismo que **lista**.  
  */  
  return ( [ elemento ] ++ lista )  
}
```

Observar que ++ recibe dos *listas*, la primera con un solo elemento

- Combinando listas literales y *append* se pueden construir otras operaciones interesantes, por ejemplo
 - `laLista_con_AgregadoAtrás`
 - que dados una `lista` y un `elemento`, lo agrega al final
 - Lo abreviamos `snoc(lista, elemento)`



```
snoc([11, 22, 44], 33)
```

es equivalente a

```
[11, 22, 44, 33]
```

```
snoc([Este, Sur], Este)
```

es equivalente a

```
[Este, Sur, Este]
```


- Combinando listas literales y *append* se pueden construir otras operaciones interesantes, por ejemplo
 - `laLista_con_AgregadoAtrás`
 - que dados una `lista` y un `elemento`, lo agrega al final
 - Lo abreviamos `snoc(lista, elemento)`

```
function snoc(lista, elemento) {  
  /* PROPÓSITO: Describir la lista resultante de agregar el  
  elemento dado al final de la lista dada.  
  PRECONDICIONES: Ninguna.  
  PARÁMETROS:  
  * lista: Una lista de cualquier tipo.  
  * elemento: Del tipo de los elementos de **lista**.  
  TIPO: El mismo que **lista**.  
  */  
  return ( lista ++ [ elemento ] )  
}
```



Aquí, en cambio, la lista de un solo elemento es la 2da

- Combinando listas literales y *append* se pueden construir otras operaciones interesantes, por ejemplo
 - `secuenciaAritméticaDeNúmerosDe_A_`
que dados dos valores, devuelva la secuencia entre ellos
 - Precisamos una repetición simple y un acumulador de listas



```
secuenciaAritméticaDeNúmerosDe_A_(1, 10)
```

es equivalente a

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
secuenciaAritméticaDeNúmerosDe_A_(40, 45)
```

es equivalente a

```
[40, 41, 42, 43, 44, 45]
```

- Combinando listas literales y *append* se pueden construir otras operaciones interesantes, por ejemplo
 - **secuenciaAritméticaDeNúmerosDe_A_**
que dados dos valores, devuelva la secuencia entre ellos
 - Precisamos una repetición simple y un acumulador de listas

```
function secuenciaAritméticaDeNúmerosDe_A_(valorInicial, valorFinal) {  
  /* PROPÓSITO: Describir la lista que contiene a todos los números  
     entre **valorInicial** y **valorFinal**, inclusive, en orden.  
     PRECONDICIONES: Ninguna  
     PARÁMETROS:      * valorInicial: Número.      * valorFinal: Número.  
     TIPO: [Número].  
     OBSERVACIÓN: Si **valorFinal** es menor a **valorInicial**  
        describe a la lista vacía.  
  */  
  próximoNúmero := valorInicial  
  listaHastaAhora := []  
  repeat (valorFinal - valorInicial + 1) {  
    listaHastaAhora := listaHastaAhora ++ [ próximoNúmero ]  
    próximoNúmero := próximoNúmero + 1  
  }  
  return (listaHastaAhora)  
}
```

La cantidad
de veces es
uno más que
la diferencia
entre los
extremos

- Combinando listas literales y *append* se pueden construir otras operaciones interesantes, por ejemplo
 - `secuenciaAritméticaDeNúmerosDe_A_`
 - que dados dos valores, devuelva la secuencia entre ellos
 - Precisamos una repetición simple y un acumulador de listas

`secuenciaAritméticaDeNúmerosDe_A_(-100, 100)` [-100, -99, -98, -97, -96, -95, -94, -93, -92, -91, -90, -89, -88, -87, -86, -85, -84, -83, -82, -81, -80, -79, -78, -77, -76, -75, -74, -73, -72, -71, -70, -69, -68, -67, -66, -65, -64, -63, -62, -61, -60, -59, -58, -57, -56, -55, -54, -53, -52, -51, -50, -49, -48, -47, -46, -45, -44, -43, -42, -41, -40, -39, -38, -37, -36, -35, -34, -33, -32, -31, -30, -29, -28, -27, -26, -25, -24, -23, -22, -21, -20, -19, -18, -17, -16, -15, -14, -13, -12, -11, -10, -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100]

es equivalente a

La expresividad de
la programación
en acción



- Las secuencias aritméticas pueden ser de otros tipos
 - ¿Cómo la definiríamos?
 - Con repetición condicional y siguiente (ejercicio...)
 - Para tipos básicos, Gobstones provee notación especial
 - `[<expValorInicial> .. <expValorFinal>]`

`[1..10]`

es equivalente a

`[1,2,3,4,5,6,7,8,9,10]`

`[40..45]`

es equivalente a

`[40,41,42,43,44,45]`

`[minColor() .. maxColor()]`

es equivalente a

`[Azul,Negro,Rojo,Verde]`

`[-100 .. 100]`

es equivalente a

`secuenciaAritméticaDeNúmerosDe_A_(-100, 100)`



- Las secuencias aritméticas pueden tener elementos a distancia mayor que uno
 - ¿Cómo la definiríamos? (Ejercicio...)
 - La sintaxis para secuencias no consecutivas es
 - $[\text{<expInicial>}, \text{<expSegundo>} \dots \text{<expFinal>}]$

$[0, 5 \dots 30]$

es equivalente a

$[0, 5, 10, 15, 20, 25, 30]$

$[10, 9 \dots 1]$

es equivalente a

$[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]$

- Combinando listas literales y *append* se pueden construir otras operaciones interesantes, por ejemplo
 - `enTotal_IgualA_`, que dadas una **cantidad** y un **elemento**, describe una lista con esa cantidad del elemento
 - Precisamos una repetición simple y un acumulador de listas



```
enTotal_IgualA_(5, Norte)
```

debe ser equivalente a

```
[ Norte, Norte, Norte, Norte, Norte ]
```

```
enTotal_IgualA_(3, 17)
```

debe ser equivalente a

```
[ 17, 17, 17 ]
```

- Combinando listas literales y *append* se pueden construir otras operaciones interesantes, por ejemplo
 - **enTotal_IgualA_**, que dadas una **cantidad** y un **elemento**, describe una lista con esa cantidad del elemento
 - Precisamos una repetición simple y un acumulador de listas

```
function enTotal_IgualA_(cantidad, elemento) {  
  /* PROPÓSITO: Describir la lista con **cantidad** elementos iguales a **elemento**.  
  PRECONDICIONES: Ninguna.  
  PARÁMETROS:  
  * cantidad: Número.  
  * elemento: Un tipo cualquiera.  
  TIPO: Una lista de elementos del mismo tipo que **elemento**.  
  OBSERVACIÓN: Si **cantidad** es menor a 0 describe a la lista vacía.  
  */  
  listaHastaAhora := []  
  repeat (cantidad) {  
    listaHastaAhora :=  
      [ elemento ] ++ listaHastaAhora  
  }  
  return (listaHastaAhora)  
}
```

Inicializar un acumulador
SOLAMENTE tiene
sentido en una
acumulación



- Combinando las operaciones definidas se pueden expresar listas de maneras más poderosas...

```
enTotal_IgualA_(5, Norte) ++ enTotal_IgualA_(4, Este)
```

es equivalente a

```
[ Norte, Norte, Norte, Norte, Norte, Este, Este, Este, Este ]
```

¿Y si fuesen muchos más?

```
enTotal_IgualA_(500, Norte) ++ enTotal_IgualA_(450, Este)
```

- Combinando listas literales y *append* se pueden construir otras operaciones interesantes, por ejemplo
 - **filaActual**
 - Precisamos la función **celdaActual** definida antes



¿Cómo sería?

- Combinando listas literales y *append* se pueden construir otras operaciones interesantes, por ejemplo
 - **filaActual**
 - Precisamos la función **celdaActual** definida antes

```
function filaActual() {  
  /* PROPÓSITO: Describir la fila actual del tablero como  
    una lista de celdas.  
    PRECONDICIONES: Ninguna.  
    TIPO: [Celda]  
  */  
  filaProcesada := []  
  IrAlBorde(Oeste)  
  while (puedeMover(Este)) {  
    filaProcesada :=  
      filaProcesada ++ [ celdaActual() ]  
    Mover(Este)  
  }  
  return (filaProcesada ++ [ celdaActual() ])  
}
```

Se agrega al final para que el orden sea el mismo que en el que fueron recorridas

- Combinando listas literales y *append* se pueden construir otras operaciones interesantes, por ejemplo
 - `tableroActual`
 - ¡Precisamos la función `filaActual` recién definida!



¿Cómo sería?



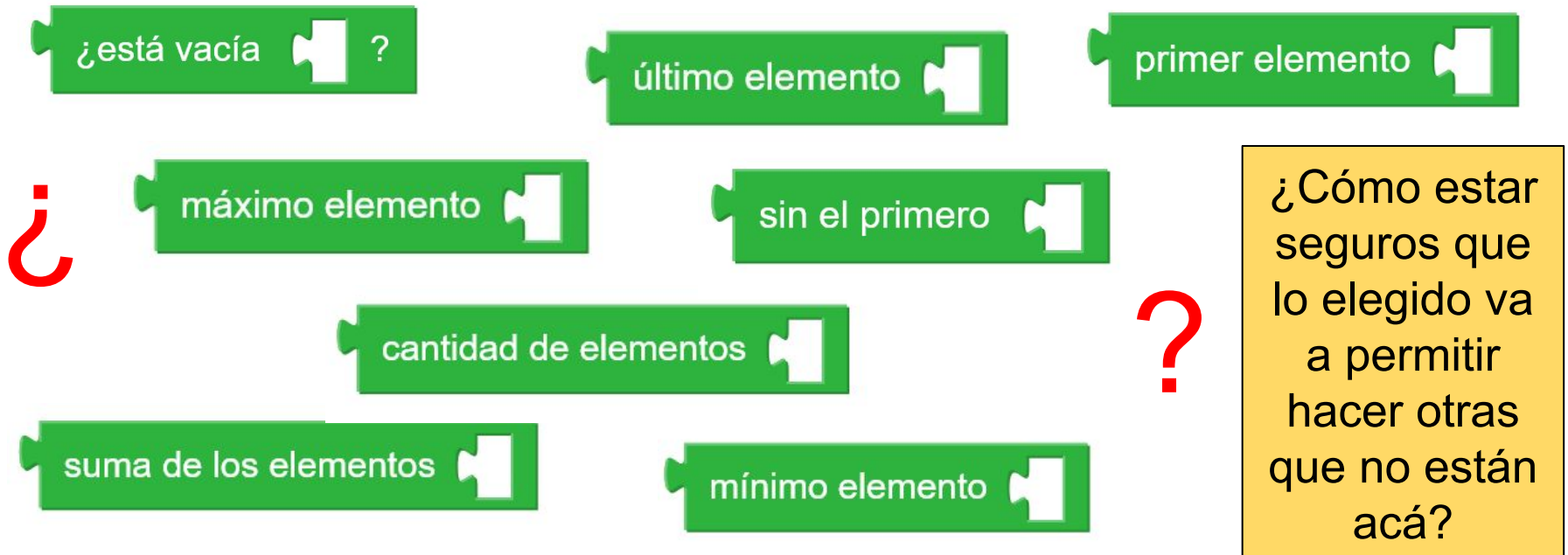
- Combinando listas literales y *append* se pueden construir otras operaciones interesantes, por ejemplo
 - **tableroActual**
 - ¡Precisamos la función **filaActual** recién definida!

```
function tableroActual() {  
  /* PROPÓSITO: Describir el tablero como una lista  
    de listas de celdas.  
    PRECONDICIONES: Ninguna.  
    TIPO: [[Celda]]  
  */  
  tableroProcesado := []  
  IrAlBorde(Norte)  
  while (puedeMover(Sur)) {  
    tableroProcesado :=  
      tableroProcesado ++ [ filaActual() ]  
    Mover(Sur)  
  }  
  return (tableroProcesado ++ [ filaActual() ])  
}
```

Se recorren las
filas y se agregan a
la lista



- ¿Cómo obtenemos información de una lista?
- Precisamos acceder a sus elementos...
 - ¡Funciones sobre listas!
 - ¿Cuál es el conjunto mínimo de expresiones primitivas que resulta más conveniente tener para construir esas funciones?





- ¿Cómo obtenemos información de una lista?
- Precisamos acceder a sus elementos...
 - ¡Funciones sobre listas!
 - ¿Cuál es el conjunto mínimo de expresiones primitivas que resulta más conveniente tener para construir esas funciones?

¿está vacía ?

último elemento

primer elemento

máximo elemento

sin el primero

cantidad de elementos

suma de los elementos

mínimo elemento

Estas son las elegidas. Veremos cómo hacer los demás



- Un conjunto adecuado de primitivas debe permitir crear todas las demás operaciones que se deseen
- El siguiente conjunto cumple esa condición
 - **primero**(*<expLista>*)
 - **resto**(*<expLista>*) (en bloques: “sinElPrimero_”)
 - **esVacía**(*<expLista>*) (en bloques: “¿estáVacía_?”)

`esVacía([10, 20, 30])`

`primero([10, 20, 30])`

¿Cuál es el contrato de cada una?

`resto([10, 20, 30])`

- **primero** (*<expLista>*)
 - **PROPÓSITO:** describir el primer elemento de la lista dada
 - **PRECONDICIÓN:** la lista dada no es vacía
 - **PARÁMETRO:** la lista es de un tipo Lista cualquiera
 - **TIPO:** el tipo de los elementos de la lista dada

```
primero([10,20,30])
```

es equivalente a

10

```
primero([Norte, Este, Sur, Este])
```

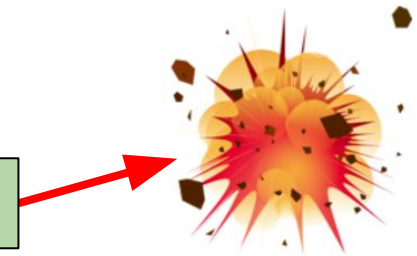
es equivalente a

Norte

Se aplica a cualquier lista, y funciona si no está vacía

```
primero([])
```

es equivalente a



BOOM

La lista no puede ser vacía.

- **resto** (*<expLista>*)
 - **PROPÓSITO:** describir una lista con los elementos de la lista dada, excepto que sin el primero de ellos
 - **PRECONDICIÓN:** la lista dada no es vacía
 - **PARÁMETRO:** la lista es de un tipo Lista cualquiera
 - **TIPO:** el tipo de la lista dada

```
resto([10,20,30])
```

es equivalente a

```
[20,30]
```

```
resto([])
```

es equivalente a



BOOM

La lista no puede ser vacía.

```
resto([Norte, Este, Sur, Este])
```

es equivalente a

```
[Este, Sur, Este]
```

Se aplica a cualquier lista, y funciona si no está vacía



- **esVacía** (*<expLista>*)
 - **PROPÓSITO:** indicar si la lista es vacía
 - **PRECONDICIÓN:** ninguna
 - **PARÁMETRO:** la lista es de un tipo Lista cualquiera
 - **TIPO:** Booleano

```
esVacía([10,20,30])
```

es Falso

Se aplica a cualquier lista

```
esVacía([Norte, Este, Sur, Este])
```


es Falso

```
esVacía([])
```

es Verdadero

- Con combinaciones de primero y resto podemos obtener cualquier elemento
 - ¡Recordar respetar los tipos!
 - O sea, es un **ERROR** escribir (`resto(primero([10,20])`), porque el argumento de `resto` debe ser una lista

```
function segundo(lista) {  
  /* PROPÓSITO: Describir el segundo elemento de la lista dada.  
  
  PRECONDICIONES: La lista dada tiene al menos dos elementos.  
  PARÁMETROS:  
  * lista: Una lista de cualquier tipo.  
  TIPO: El de los elementos de la lista dada.  
  */  
  return (primero(resto(lista)))  
}
```



El segundo queda primero
luego de sacar el primero



- Con combinaciones de primero y resto podemos obtener cualquier elemento
 - ¡Recordar respetar los tipos!

```
function sinLosDosPrimeros(lista) {  
  /* PROPÓSITO: Describir la lista resultante de quitarle a  
    la lista dada sus primeros dos elementos.  
    PRECONDICIONES: La lista dada tiene al menos dos elementos.  
    PARÁMETROS:  
    * lista: Una lista de cualquier tipo.  
    TIPO: El mismo que la lista dada.  
  */  
  return (resto(resto(lista)))  
}
```

Al sacar el primero 2 veces seguidas, hay 2 elementos menos



- Con combinaciones de primero y resto, podemos hacer muchas otras operaciones
 - Contar, sumar o modificar elementos
 - Buscar, elegir, eliminar o agregar elementos
 - Todos implican **recorrer** la lista de a un elemento por vez

cantidad de elementos

máximo elemento

suma de los elementos

Éstas son solo algunas de las que podemos hacer

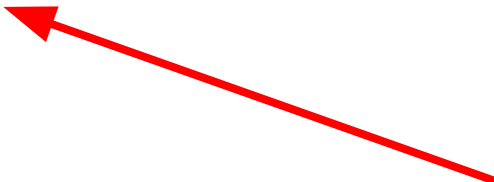
mínimo elemento

último elemento



- Con combinaciones de primero y resto podemos obtener cualquier elemento
 - ¡Recordar respetar los tipos!

```
function tercero(lista) {  
  /* PROPÓSITO: Describir el tercer elemento de la lista dada.  
   PRECONDICIONES: La lista dada tiene al menos tres elementos.  
   PARÁMETROS:  
   * lista: Una lista de cualquier tipo.  
   TIPO: El de los elementos de la lista dada.  
  */  
  return (primero(sinLosDosPrimeros(lista)))  
}
```



Y al sacar 2, el que queda primero es el tercero

- Ejercicio:
 - construir una función que describa un mazo
`mazoDeCartasEspañolas`
 - Precisamos la función `paloSiguiente_` definida antes





Cierre



Cierre



- **Listas**

- Son datos con estructura
- Tienen muchas partes, pero no siempre la misma cantidad
- Se pueden crear a través de funciones constructoras
- Se puede obtener información de ellas a través de funciones de acceso
- Se pueden hacer recorridos sobre los elementos de una lista
- Son un tipo de datos muy poderoso y útil