



Introducción a la Programación

Clases teóricas

por Pablo E. “Fidel” Martínez López

11. Procesamiento de listas





Repaso



- **Programar es comunicar** (con máquinas y personas)
 - Estrategia de solución (división en subtarefas)
 - Legibilidad (elección de nombres, indentación)
 - **CONTRATOS:** Propósito, parámetros y precondiciones
- **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
- Alternativa condicional
- Repetición condicional
- Asignación de variables



● Expresiones

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



- **Tipos de datos**

- Básicos

- Colores, Direcciones, Números, Booleanos

- Con estructura

- Registros (muchas partes, diferentes tipos, cantidad fija)
 - Variantes (una sola parte, muchas posibilidades)
 - **Listas** (muchos elementos, mismo tipo, cantidad variable)



Procesamiento de listas



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

cantidad de elementos

máximo elemento

suma de los elementos

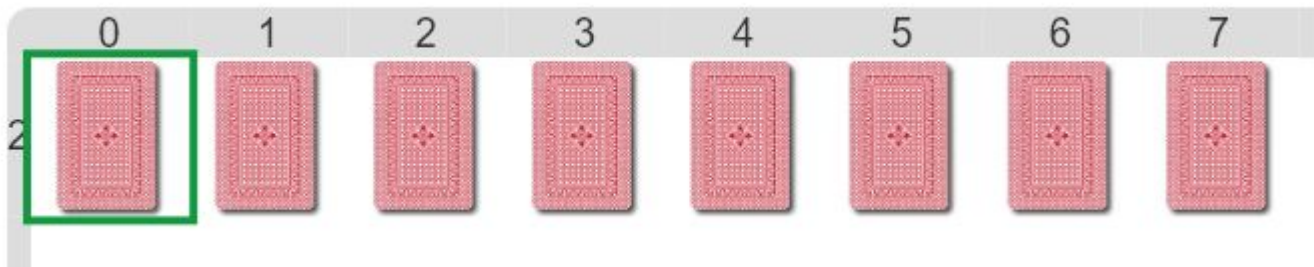
Éstas son solo algunas de las que podemos hacer

mínimo elemento

último elemento

- Recordemos cómo es la estructura de un **recorrido**

```
IniciarRecorrido()  
while (quedanElementos()) {  
    ProcesarElemento()  
    PasarAlSiguienteElemento()  
}  
FinalizarRecorrido()
```



Los elementos pueden estar en el tablero...
...¡o accederse desde una lista!



- Por ejemplo, contar cuántos elementos hay en la lista
 - ¡Se recorre, y en cada paso, se cuenta el actual!



8

Dada una lista de entrada, debe describir un número



- Por ejemplo, contar cuántos elementos hay en la lista
 - ¡Se recorre, y en cada paso, se cuenta el actual!

```
function cantidadDeElementosEn_(lista) {  
  /* PROPÓSITO: Describir la cantidad de elementos en la  
    lista dada.  
    PRECONDICIONES: Ninguna.  
    PARÁMETROS:  
    * lista: Una lista de cualquier tipo.  
    TIPO: Número.  
    OBSERVACIÓN: Es un recorrido de acumulación sobre la  
    lista dada.  
  */  
  ...  
}
```

¡PRIMERO EL CONTRATO!



- Por ejemplo, contar cuántos elementos hay en la lista
 - ¡Se recorre, y en cada paso, se cuenta el actual!

```
function cantidadDeElementosEn_(lista) {  
    /* ... */
```

```
    IniciarRecorrido (recordar que faltan todos y no conté nada)
```

```
    while quedanElementos (la lista de los que faltan no está vacía) {
```

```
        ProcesarElementoActual (contar el primero de los que faltan)
```

```
        PasarAlSiguienteElemento (recordar que saqué el primero)
```

```
    }
```

```
    FinalizarRecorrido (describir el resultado final)
```

```
}
```



OBSERVEMOS LA ESTRUCTURA
DE RECORRIDO



- Por ejemplo, contar cuántos elementos hay en la lista
 - ¡Se recorre, y en cada paso, se cuenta el actual!

```
function cantidadDeElementosEn_(lista) {  
  /* ... */  
  listaRestante := lista      // Al principio faltan todos  
  cantidadContada := 0        // y no conté ninguno  
  while quedanElementos (la lista de los que faltan no está vacía) {  
    ProcesarElementoActual (contar el primero de los que faltan)  
    PasarAlSiguienteElemento (recordar que saqué el primero)  
  }  
  FinalizarRecorrido (describir el resultado final)  
}
```



OBSERVEMOS LA ESTRUCTURA
DE RECORRIDO



- Por ejemplo, contar cuántos elementos hay en la lista
 - ¡Se recorre, y en cada paso, se cuenta el actual!

```
function cantidadDeElementosEn_(lista) {  
  /* ... */  
  listaRestante := lista      // Al principio faltan todos  
  cantidadContada := 0        // y no conté ninguno  
  while (not esVacía(listaRestante))  
  {  
    ProcesarElementoActual (contar el primero de los que faltan)  
    PasarAlSiguienteElemento (recordar que saqué el primero)  
  }  
  FinalizarRecorrido (describir el resultado final)  
}
```



OBSERVEMOS LA ESTRUCTURA
DE RECORRIDO



- Por ejemplo, contar cuántos elementos hay en la lista
 - ¡Se recorre, y en cada paso, se cuenta el actual!

```
function cantidadDeElementosEn_(lista) {  
  /* ... */  
  listaRestante := lista      // Al principio faltan todos  
  cantidadContada := 0        // y no conté ninguno  
  while (not esVacía(listaRestante)) {  
    cantidadContada := cantidadContada + 1 // Cuento uno  
    PasarAlSiguienteElemento (recordar que saqué el primero)  
  }  
  FinalizarRecorrido (describir el resultado final)  
}
```



OBSERVEMOS LA ESTRUCTURA
DE RECORRIDO



- Por ejemplo, contar cuántos elementos hay en la lista
 - ¡Se recorre, y en cada paso, se cuenta el actual!

```
function cantidadDeElementosEn_(lista) {  
  /* ... */  
  listaRestante := lista      // Al principio faltan todos  
  cantidadContada := 0        // y no conté ninguno  
  while (not esVacía(listaRestante)) {  
    cantidadContada := cantidadContada + 1 // Cuento uno  
    listaRestante := resto(listaRestante)  // y lo saco  
  }  
  FinalizarRecorrido (describir el resultado final)  
}
```



OBSERVEMOS LA ESTRUCTURA
DE RECORRIDO



- Por ejemplo, contar cuántos elementos hay en la lista
 - ¡Se recorre, y en cada paso, se cuenta el actual!

```
function cantidadDeElementosEn_(lista) {  
  /* ... */  
  listaRestante := lista      // Al principio faltan todos  
  cantidadContada := 0        // y no conté ninguno  
  while (not esVacía(listaRestante)) {  
    cantidadContada := cantidadContada + 1 // Cuento uno  
    listaRestante := resto(listaRestante)  // y lo saco  
  }  
  return (cantidadContada)  
}
```



OBSERVEMOS LA ESTRUCTURA
DE RECORRIDO



- Por ejemplo, encontrar el mínimo elemento
 - Se recorre, y en cada paso, se elige el más chico visto hasta el momento
 - Hay que establecer cuál es el orden a considerar

[12, 7, 1, 8, 6]



1

Dada una lista,
describir el elemento
más chico



- Por ejemplo, encontrar el mínimo elemento
 - Se recorre, y en cada paso, se elige el más chico

¡Siempre primero el contrato!

```
function mínimoElementoDe_(lista) {  
  /* PROPÓSITO: Describir el menor elemento de la lista dada.  
  PRECONDICIONES: La lista dada no es vacía.  
  PARÁMETROS:  
  * lista: Una lista de cualquier tipo básico.  
  TIPO: El de los elementos de la lista dada.  
  OBSERVACIÓN: Es un recorrido de mínimo/máximo sobre la  
    lista dada.  
  */  
  ...  
}
```

En este caso, la lista no puede estar vacía...



- Por ejemplo, encontrar el mínimo elemento
 - Se recorre, y en cada paso, se elige el más chico

```
function mínimoElementoDe_(lista) {  
  /* ... */
```

```
  IniciarProcesamiento (recordar que el mínimo visto es el 1°)
```

```
  IniciarRecorrido (recordar que falta procesar los demás )
```

```
  while quedanElementos (la lista de los que faltan no está vacía) {
```

```
    ProcesarElementoActual (ver si el mínimo se mantiene o no)
```

```
    PasarAlSiguienteElemento (recordar que saqué el primero)
```

```
  }
```

```
  FinalizarRecorrido (describir el resultado final)
```

```
}
```

Observar la estructura de recorrido



- Por ejemplo, encontrar el mínimo elemento
 - Se recorre, y en cada paso, se elige el más chico

```
function mínimoElementoDe_(lista) {  
  /* ... */  
  mínimoVisto := primero(lista) // El 1° es el mínimo por ahora  
  IniciarRecorrido (recordar que falta procesar los demás )  
  while quedanElementos (la lista de los que faltan no está vacía) {  
    ProcesarElementoActual (ver si el mínimo se mantiene o no)  
    PasarAlSiguienteElemento (recordar que saqué el primero)  
  }  
  FinalizarRecorrido (describir el resultado final)  
}
```

¿Por qué empezar con el primero de la lista?
Porque si está vacía, no hay mínimo



- Por ejemplo, encontrar el mínimo elemento
 - Se recorre, y en cada paso, se elige el más chico

```
function mínimoElementoDe_(lista) {  
  /* ... */  
  mínimoVisto := primero(lista) // El 1° es el mínimo por ahora  
  listaRestante := resto(lista)  // y faltan mirar los demás  
  while quedanElementos (la lista de los que faltan no está vacía) {  
    ProcesarElementoActual (ver si el mínimo se mantiene o no)  
    PasarAlSiguienteElemento (recordar que saqué el primero)  
  }  
  FinalizarRecorrido (describir el resultado final)  
}
```

Si ya miré el primero, faltan procesar los demás



- Por ejemplo, encontrar el mínimo elemento
 - Se recorre, y en cada paso, se elige el más chico

```
function mínimoElementoDe_(lista) {  
  /* ... */  
  mínimoVisto := primero(lista) // El 1° es el mínimo por ahora  
  listaRestante := resto(lista)  // y faltan mirar los demás  
  while (not esVacía(listaRestante)) {  
    ProcesarElementoActual (ver si el mínimo se mantiene o no)  
    PasarAlSiguienteElemento (recordar que saqué el primero)  
  }  
  FinalizarRecorrido (describir el resultado final)  
}
```

Termina cuando no hay más elementos para mirar



- Por ejemplo, encontrar el mínimo elemento
 - Se recorre, y en cada paso, se elige el más chico

```
function mínimoElementoDe_(lista) {  
  /* ... */  
  mínimoVisto := primero(lista) // El 1° es el mínimo por ahora  
  listaRestante := resto(lista)  // y faltan mirar los demás  
  while (not esVacía(listaRestante)) {  
    // Comparo el mínimo hasta ahora con el primero de los  
    mínimoVisto := // que faltan  
      mínimoEntre_y_(mínimoVisto, primero(listaRestante))  
    PasarAlSiguienteElemento (recordar que saqué el primero)  
  }  
  FinalizarRecorrido (describir el resultado final)  
}
```

El procesamiento elige si cambiar el mínimo visto o no



- Por ejemplo, encontrar el mínimo elemento
 - Se recorre, y en cada paso, se elige el más chico

```
function mínimoElementoDe_(lista) {  
  /* ... */  
  mínimoVisto := primero(lista) // El 1° es el mínimo por ahora  
  listaRestante := resto(lista)  // y faltan mirar los demás  
  while (not esVacía(listaRestante)) {  
    // Comparo el mínimo hasta ahora con el primero de los  
    mínimoVisto :=                                // que faltan  
      mínimoEntre_y_(mínimoVisto, primero(listaRestante))  
    listaRestante := resto(listaRestante) // y lo saco  
  }  
  FinalizarRecorrido (describir el resultado final)  
}
```

Pasar el siguiente, como siempre, quita el
procesado de la lista de los que faltan



- Por ejemplo, encontrar el mínimo elemento
 - Se recorre, y en cada paso, se elige el más chico

```
function mínimoElementoDe_(lista) {  
    /* ... */  
    mínimoVisto := primero(lista) // El 1° es el mínimo por ahora  
    listaRestante := resto(lista)  // y faltan mirar los demás  
    while (not esVacía(listaRestante)) {  
        // Comparo el mínimo hasta ahora con el primero de los  
        mínimoVisto := // que faltan  
            mínimoEntre_y_(mínimoVisto, primero(listaRestante))  
        listaRestante := resto(listaRestante) // y lo saco  
    }  
    return (mínimoVisto)  
}
```

Al terminar, el mínimo visto es el mínimo de toda la lista



- Por ejemplo, encontrar el mínimo elemento
 - Se recorre, y en cada paso, se elige el más chico

Para elegir el mínimo se usa una alternativa condicional

```
function mínimoEntre_y_(elemento1, elemento2) {  
  /* PROPÓSITO: Describir el mínimo entre los dos elementos  
     dados. Si son iguales, describe cualquiera de ellos.  
     PRECONDICIONES: Ninguna.  
     PARÁMETROS:  
     * elemento1: De un tipo básico cualquiera.  
     * elemento2: Del mismo tipo que **elemento1**.  
     TIPO: El mismo que **elemento1**.  
  */  
  return (choose elemento1 when      (elemento1 < elemento2)  
           elemento2 otherwise // (elemento1 >= elemento2)  
  )  
}
```



- Por ejemplo, transformar una lista
 - Se recorre, y en cada paso, se transforma el actual y se lo agrega al resultado

Dada una lista de cartas, describir la lista de sus números



2 11 12 4 6 10 1 3 5 5



- Por ejemplo, transformar una lista
 - Se recorre, y en cada paso, se transforma el actual

```
function númerosDeLasCartasDe_(mazo) {  
  /* PROPÓSITO: Describir la lista de números de las  
    cartas del mazo dado.  
    PRECONDICIONES: Ninguna.  
    PARÁMETROS:  
    * mazo: [Carta].  
    TIPO: [Número].  
    OBSERVACIÓN: Es un recorrido de transformación  
    sobre el mazo dado (una lista).  
  */  
  ...  
}
```

En este caso, hay que
armar una lista
resultado

¡¡NO OLVIDAR!!
PRIMERO EL CONTRATO



- Por ejemplo, transformar una lista
 - Se recorre, y en cada paso, se transforma el actual

```
function númerosDeLasCartasDe_(mazo) {  
    /* ... */
```

Es un recorrido de
listas

IniciarRecorrido (recordar que faltan todos)

IniciarAcumulación (recordar que no transformé ninguno)

```
while quedanElementos (la lista de los que faltan no está vacía) {
```

ProcesarElementoActual (transformar el primero)

PasarAlSiguienteElemento (recordar que saqué el primero)

```
}
```

FinalizarRecorrido (describir el resultado final)

```
}
```



- Por ejemplo, transformar una lista
 - Se recorre, y en cada paso, se transforma el actual

```
function númerosDeLasCartasDe_(mazo) {  
  /* ... */  
  mazoRestante := mazo // Al principio faltan todas  
  IniciarAcumulación (recordar que no transformé ninguno)  
  while (not esVacía(mazoRestante)) {  
    ProcesarElementoActual (transformar el primero)  
    // Saco la primera de entre las que me quedan  
    mazoRestante := resto(mazoRestante)  
  }  
  FinalizarRecorrido (describir el resultado final)  
}
```

Es un recorrido de listas



- Por ejemplo, transformar una lista
 - Se recorre, y en cada paso, se transforma el actual

```
function númerosDeLasCartasDe_(mazo) {  
  /* ... */
```

El acumulador es
una lista

```
  IniciarRecorrido (recordar que faltan todos)
```

```
  númerosVistos := [] // y no ví ninguno
```

```
  while quedanElementos (la lista de los que faltan no está vacía) {
```

```
    númerosVistos := númerosVistos ++  
                        [ número(primer(mazoRestante)) ]
```

```
    PasarAlSiguienteElemento (recordar que saqué el primero)
```

```
  }
```

```
  FinalizarRecorrido (describir el resultado final)
```

```
}
```



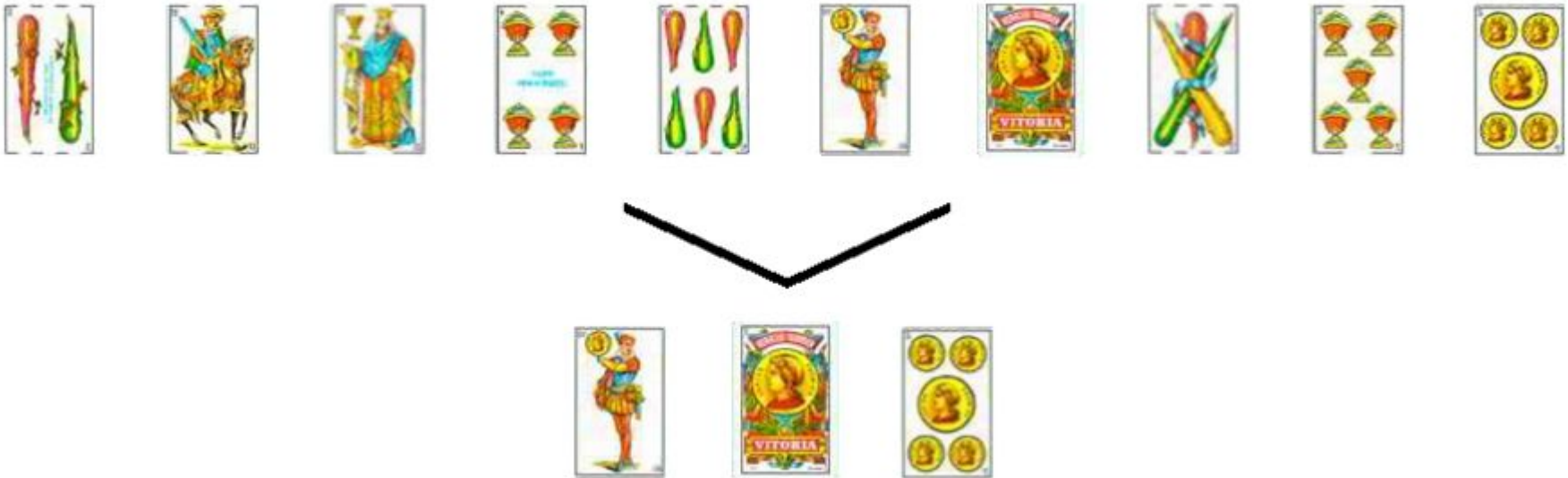

- Por ejemplo, transformar una lista
 - Se recorre, y en cada paso, se transforma el actual

```
function númerosDeLasCartasDe_(mazo)
/* ... */
mazoRestante := mazo // Al principio faltan todas
númerosVistos := []   // y no ví ninguno
while (not esVacía(mazoRestante))
    númerosVistos := númerosVistos ++ [ número(primer(mazoRestante)) ]
    // Saco la primera de entre las que me quedan
    mazoRestante := resto(mazoRestante)
}
return (númerosVistos)
}
```

Procesar es agregar el número de la 1er carta



- Por ejemplo, eliminar elementos que no queremos
 - Se recorre, y en cada paso, se decide si el elemento actual se deja o se quita



Dada una lista de cartas, quedarse solamente con las de Oros



- Por ejemplo, eliminar elementos que no queremos
 - Se recorre, y en cada paso, se decide si dejarlo o no

```
function soloLosOrosDe_(mazo) {  
  /* PROPÓSITO: Describir la lista de cartas  
    del mazo dado que son de Oros.  
    PRECONDICIONES: Ninguna.  
    PARÁMETROS:  
    * mazo: [Carta].  
    TIPO: [Carta].  
    OBSERVACIÓN: Es un recorrido de filtro sobre  
    el mazo dado (una lista).  
  */  
  ...  
}
```

Siempre, siempre,
primero el contrato



- Por ejemplo, eliminar elementos que no queremos
 - Se recorre, y en cada paso, se decide si dejarlo o no

```
function soloLosOrosDe_(mazo) {  
  /* ... */  
  mazoRestante := mazo           // Al principio faltan todas  
  mazoDeOrosVistos := []         // y no ví ninguna  
  while (not esVacía(mazoRestante)) {  
    // Agrego la carta, solo si es de Oros  
    mazoDeOrosVistos := mazoDeOrosVistos ++  
      singular_Si_(primero(mazoRestante),  
                    esDeOros_(primero(mazoRestante)))  
    // Saco la primera de entre las que me quedan  
    mazoRestante := resto(mazoRestante)  
  }  
  return (mazoDeOrosVistos)  
}
```

La lista que se agrega puede tener un elemento o ninguno, según la condición

¡Nuevamente, un recorrido!



- Por ejemplo, eliminar elementos que no queremos
 - Se recorre, y en cada paso, se decide si dejarlo o no

La lista resultado puede tener un elemento o ninguno, según la condición

```
function singular_Si_(elemento, condición) {  
  /* PROPÓSITO: Describir una lista según el valor de la  
    condición dada. Si es verdadera, describe la lista singular  
    con **elemento**. Si no, describe la lista vacía.  
    PRECONDICIONES: Ninguna.  
    PARÁMETROS:  
    * elemento: De un tipo cualquiera.  
    * condición: Booleano.  
    TIPO: Lista del tipo de **elemento**.  
  */  
  return (choose [elemento] when (condición)  
              []      otherwise)  
}
```



- Por ejemplo, buscar un elemento
 - Se recorre, y si encuentra lo buscado, termina (y si no, termina cuando no hay más elementos)



Dada una Lista de Cartas, describir si está el ancho de Espadas

SI



NO



- Por ejemplo, buscar un elemento
 - Se recorre, y si encuentra lo buscado, termina

¿Qué debemos hacer primero?



- Por ejemplo, buscar un elemento
 - Se recorre, y si encuentra lo buscado, termina

```
function estáElAnchoDeEspadasEn_(mazo) {  
  /* PROPÓSITO: Indicar si el mazo dado contiene  
    algún ancho de Espadas.  
    PRECONDICIONES: Ninguna.  
    PARÁMETROS:  
    * mazo: [Carta].  
    TIPO: Booleano.  
    OBSERVACIÓN: Es un recorrido de búsqueda  
    sobre el mazo dado (una lista).  
  */  
  ...  
}
```

¡Sí! ¡EL CONTRATO!



- Por ejemplo, buscar un elemento
 - Se recorre, y si encuentra lo buscado, termina

```
function estáElAnchoDeEspadasEn_(mazo) {  
    /* ... */  
    IniciarRecorrido (recordar que faltan todos)  
    while ( quedanElementos (la lista de los que faltan no está vacía)  
            y no encontréLoBuscado (la primera no es el ancho)  
        ) {  
        PasarAlSiguienteElemento (recordar que saqué el primero)  
    }  
    FinalizarRecorrido (describir el resultado final)  
}
```

¡Este recorrido puede terminar antes!



- Por ejemplo, buscar un elemento
 - Se recorre, y si encuentra lo buscado, termina

```
function estáElAnchoDeEspadasEn_(mazo) {  
    /* ... */  
    mazoRestante := mazo  
    while ( quedanElementos (la lista de los que faltan no está vacía)  
           y no encontréLoBuscado (la primera no es el ancho)  
        ) {  
        mazoRestante := resto(mazoRestante)  
    }  
    FinalizarRecorrido (describir el resultado final)  
}
```

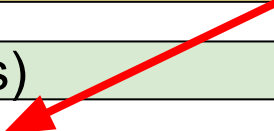
Recorrer una lista es como siempre



- Por ejemplo, buscar un elemento
 - Se recorre, y si encuentra lo buscado, termina

```
function estáElAnchoDeEspadasEn_(mazo) {  
    /* ... */  
    IniciarRecorrido (recordar que faltan todos)  
    while ( not esVacía(mazoRestante) &&  
           not esAnchoDeEspadas_(primero(mazoRestante))  
           ) {  
        PasarAlSiguienteElemento (recordar que saqué el primero)  
    }  
    FinalizarRecorrido (describir el resultado final)  
}
```

La condición de fin
requiere circuito corto



¿Cómo saber si encontré lo buscado?



- Por ejemplo, buscar un elemento
 - Se recorre, y si encuentra lo buscado, termina

```
function estáElAnchoDeEspadasEn_(mazo) {  
    /* ... */  
    mazoRestante := mazo  
    while ( not esVacía(mazoRestante) &&  
            not esAnchoDeEspadas_(primero(mazoRestante))  
        ) {  
        mazoRestante := resto(mazoRestante)  
    }  
    // Si quedan cartas, es porque encontré lo que buscaba.  
    return (not esVacía(mazoRestante))  
}
```

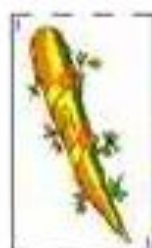
Si encontré lo que buscaba, tienen que quedar cartas



Procesamiento modularizado



- ¿Cómo hacer si quiero eliminar una carta?
 - Hay una sola que cumple la condición



Dar el mazo, sin el ancho de espadas



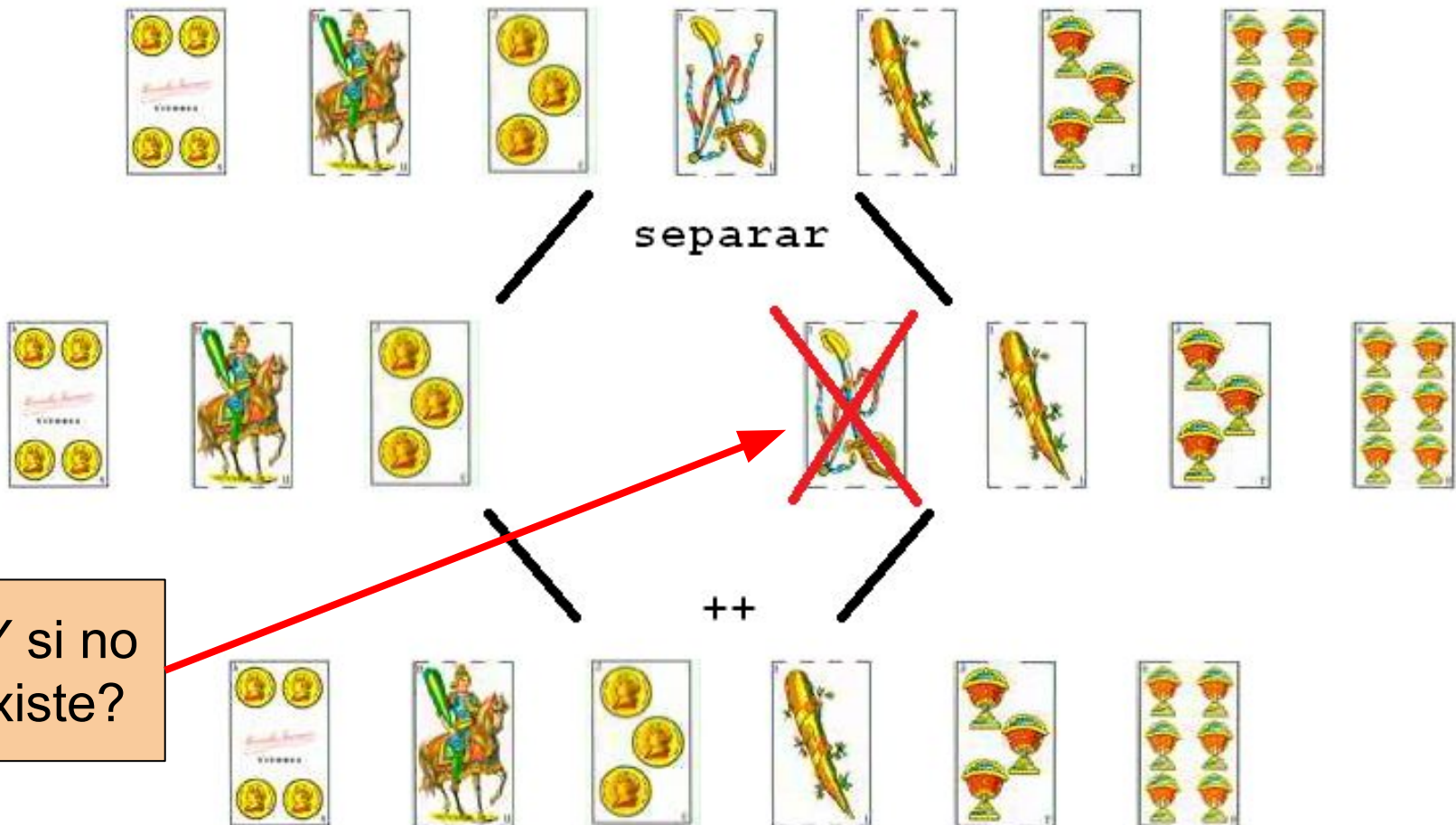
- ¿Cómo hacer si quiero eliminar una carta?
 - Hay una sola que cumple la condición

¿Cómo hacerla
sin recorrer TODA
la lista?

```
function sinElAnchoDeEspadas_(mazo) {  
  /* PROPÓSITO: Describir al mazo resultante de quitar  
    el ancho de Espadas del mazo dado.  
    PRECONDICIONES: El mazo dado contiene a lo sumo  
    UN ancho de Espadas.  
    PARÁMETROS:  
    * mazo: [Carta].  
    TIPO: [Carta].  
    OBSERVACIÓN: Usa la idea de separar la lista en  
    dos partes.  
  */  
  ...  
}
```



- ¿Cómo hacer si quiero eliminar una carta?
 - Hay una sola que cumple la condición





- ¿Cómo hacer si quiero eliminar una carta?
 - Hay una sola que cumple la condición

```
function sinElAnchoDeEspadas_(mazo) {  
  /* PROPÓSITO: Describir al mazo resultante de quitar el ancho de  
    Espadas del mazo dado.  
    PRECONDICIONES: El mazo dado contiene a lo sumo UN ancho de  
    Espadas.  
    PARÁMETROS: * mazo: [Carta].  
    TIPO: [Carta].  
    OBSERVACIÓN: Usa la idea de separar la lista en dos partes.  
  */  
  return (choose cartasAntesDelAnchoDeEspadasEn_(mazo) ++  
           resto(cartasDesdeElAnchoDeEspadasEn_(mazo))  
           when (estáElAnchoDeEspadasEn_(mazo))  
               mazo otherwise)  
}
```

¡Ojo con la parcialidad
en el caso de borde!



- ¿Y cómo *separar* una lista en 2 partes?
 - El lugar de corte lo indica una condición



Separar en 2 partes,
cortando en el ancho
de espadas



- Se puede *separar* una lista en 2 partes
 - El lugar de corte lo indica una condición

```
function cartasAntesDelAnchoDeEspadasEn_(mazo) {  
  /* PROPÓSITO: Describir la lista con las cartas del mazo dado que  
    están antes del ancho de Espadas.  
    PRECONDICIONES: El ancho de Espadas está en el mazo.  
    PARÁMETROS: * mazo: [Carta].  
    TIPO: [Carta].  
    OBSERVACIÓN: Es un recorrido de búsqueda  
      con acumulación sobre el mazo dado.  
  */  
  ...  
}  
  
function cartasDesdeElAnchoDeEspadasEn_(mazo) {  
  /* PROPÓSITO: Describir la lista con las cartas del mazo dado que  
    están a partir del ancho de Espadas.  
    PRECONDICIONES: El ancho de Espadas está en el mazo.  
    PARÁMETROS: * mazo: [Carta].  
    TIPO: [Carta].  
    OBSERVACIÓN: Es un recorrido de búsqueda sobre el mazo dado.  
  */  
  ...  
}
```

¿Sabías que
los contratos
van primero?
;)



- Se puede *separar* una lista en 2 partes (parte 1)
 - El lugar de corte lo indica una condición

```
function cartasDesdeElAnchoDeEspadasEn_(mazo) {  
  /* ... */  
  mazoRestante := mazo  
  while (not esAnchoDeEspadas_(primero(mazoRestante))) {  
    mazoRestante := resto(mazoRestante)  
  }  
  return (mazoRestante)  
}
```

Primero describimos
la “mitad” de atrás

Un recorrido de búsqueda
que retorna una lista



- Se puede *separar* una lista en 2 partes (parte 2)
 - El lugar de corte lo indica una condición

```
function cartasAntesDelAnchoDeEspadasEn_(mazo) {  
  /* ... */  
  mazoRestante := mazo  
  cartasVistas := []  
  while (not esAnchoDeEspadas_(primero(mazoRestante))) {  
    cartasVistas := cartasVistas ++ [ primero(mazoRestante) ]  
    mazoRestante := resto(mazoRestante)  
  }  
  return (cartasVistas)  
}
```

Un recorrido de
búsqueda con
acumulación

Y luego, la “mitad” de adelante



- La separación se puede usar para diversas cosas
 - Por ejemplo, ver si está un elemento (o sacarlo)

```
function estáElAnchoDeEspadasEn_Modular(mazo) {  
  /* PROPÓSITO: Indicar si el mazo dado contiene  
    algún ancho de Espadas.  
    PRECONDICIONES: Ninguna.  
    PARÁMETROS:  
    * mazo: [Carta].  
    TIPO: Booleano.  
    OBSERVACIÓN: Resuelve utilizando la lista  
      a partir del ancho.  
  */  
  return (not esVacía(cartasDesdeElAnchoDeEspadasEn_(mazo)))  
}
```




La operación de búsqueda la hace la subtaska



- Los recorridos que recorren toda la lista, siguen siempre el mismo esquema de trabajo
 - ¿Será posible expresar este esquema con una herramienta?

```
function montoAPagarPor_(carritoDeCompras) {  
  /* PROPÓSITO: Describir la suma de todos los precios  
    de los productos en el carrito dado.  
    PRECONDICIONES: Ninguna.  
    PARÁMETROS:  
    * carritoDeCompras: [Producto].  
    TIPO: Número.  
    OBSERVACIÓN: Es un recorrido de acumulación sobre la lista  
    dada.  
  */  
  ...  
}
```



Se debe hacer un recorrido para
sumar los precios de todos los
productos



- Los recorridos que recorren toda la lista, siguen siempre el mismo esquema de trabajo
 - ¿Será posible expresar este esquema con una herramienta?

```
function montoAPagarPor_(carritoDeCompras) {  
  /* ... */  
  montoHastaAhora := 0  
  productosRestantes := carritoDeCompras  
  while (not esVacía(productosRestantes)) {  
    montoHastaAhora := montoHastaAhora +  
                        precio(primeros(productosRestantes))  
  
    productosRestantes := resto(productosRestantes)  
  }  
  return (montoHastaAhora)  
}
```




- Los recorridos que recorren toda la lista, siguen siempre el mismo esquema de trabajo
 - ¿Será posible expresar este esquema con una herramienta?

```
function montoAPagarPor_(carritoDeCompras) {  
  /* ... */  
  montoHastaAhora := 0  
  IniciarRecorrido (recordar que faltan todos)  
  while quedanElementos (la lista de los que faltan no está vacía) {  
    montoHastaAhora := montoHastaAhora +  
      precio(primerO(productosRestantes))  
    PasarAlSiguienteElemento (recordar que saqué el primero)  
  }  
  return (montoHastaAhora)  
}
```

¡Estas operaciones son siempre iguales!

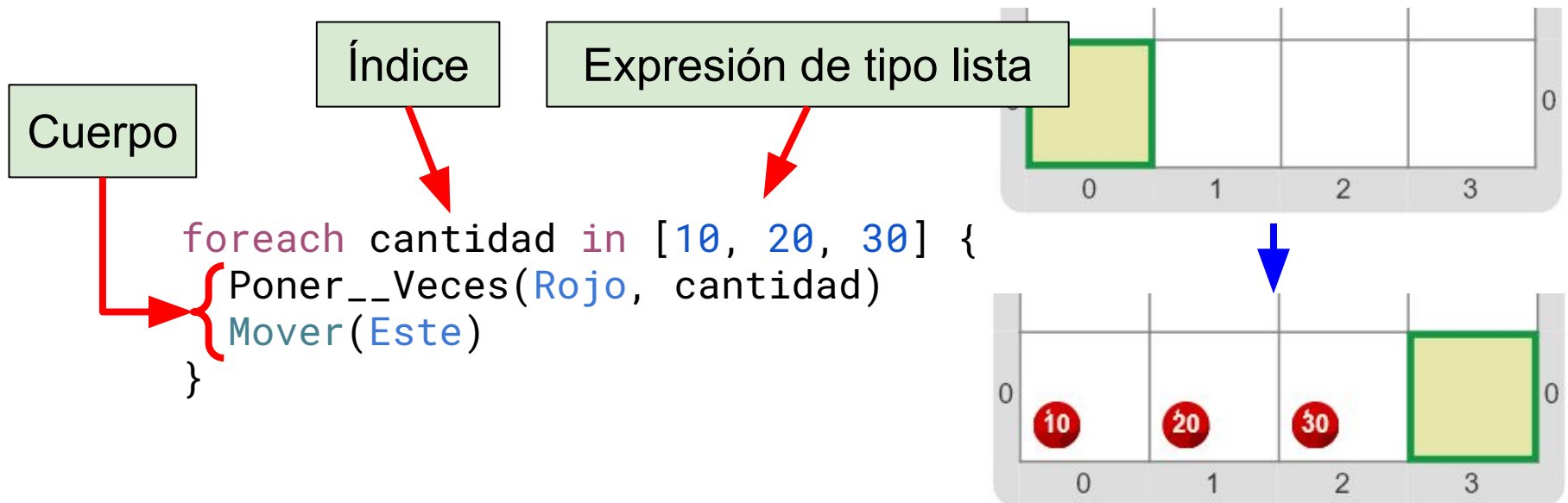


- Los recorridos se pueden expresar con una operación primitiva (siempre que NO sean de búsqueda)
 - Se llama **repetición indexada**
 - La palabra clave es **foreach** (para cada uno) y podemos leerla como “recorrer cada”

```
function montoAPagarPor_Indexada(carritoDeCompras) {  
  /* ... */  
  montoHastaAhora := 0  
  foreach producto in carritoDeCompras {  
    montoHastaAhora := montoHastaAhora + precio(producto)  
  }  
  return (montoHastaAhora)  
}
```

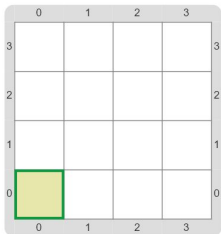
El índice toma el valor del elemento actual en cada repetición

- La repetición indexada usa la palabra clave **foreach**
 - Tiene 3 partes: un índice, una lista y un cuerpo
foreach *<nombreÍndice>* **in** *<expresiónDeLista>*
<bloqueCuerpo>
 - Ejecuta el bloque por cada elemento de la lista
 - El índice toma el valor de cada elemento por turno





- La expresión del foreach puede ser cualquiera, siempre que tenga tipo lista
- El índice es un nombre con minúsculas que se puede usar en el cuerpo, y toma valores en esa lista



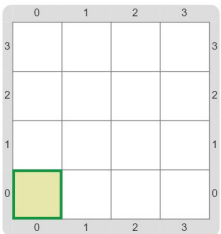
```
Poner(Verde)
foreach dir in (con_veces_(3,Norte)
               ++con_veces_(2,Este)++[Sur]) {
    Mover(dir)
    Poner(Verde)
}
```



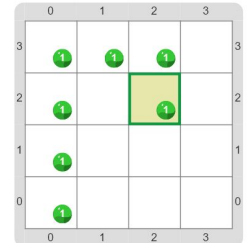
```
function con_veces_(cantidad,elemento) {
    /*...*/
    listaArmadaHastaAhora := []
    repeat(cantidad) {
        listaArmadaHastaAhora :=
            listaArmadaHastaAhora ++ [elemento]
    }
    return(listaArmadaHastaAhora)
}
```



- La expresión del foreach puede ser cualquiera, siempre que tenga tipo lista
- El índice es un nombre con minúsculas que se puede usar en el cuerpo, y toma valores en esa lista



```
Poner(Verde)
foreach dir in (con_veces_(3,Norte)
               ++con_veces_(2,Este)++[Sur]) {
  Mover(dir)
  Poner(Verde)
}
```



Uso del índice

Expresión de tipo lista

```
function con_veces_(cantidad,elemento) {
  /*...*/
  listaArmadaHastaAhora := []
  repeat(cantidad) {
    listaArmadaHastaAhora :=
      listaArmadaHastaAhora ++ [elemento]
  }
  return(listaArmadaHastaAhora)
}
```



Cierre



Cierre

- ***Procesamiento de listas***

- Para procesar listas usamos la estructura de recorrido
 - Usando **primero**, **esVacía** y **resto**
- Se pueden obtener distintos procesamientos con la misma estructura
 - Cálculo de totales (cantidad, suma, mínimo, etc.)
 - Transformación de listas (cartas a números, etc.)
 - Eliminación de elementos (filtros)
 - Búsquedas (frenando antes del final)
 - Separación en 2 partes

- La ***repetición indexada*** sirve para recorrer listas (pero NO recorridos de búsqueda)