



Introducción a la Programación

Clases teóricas

por Pablo E. “Fidel” Martínez López

10. 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
- **Alternativa indexada**



● 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
- Alternativa indexada en expresiones



- **Tipos de datos**

- Básicos

- Colores, Direcciones, Números, Booleanos

- 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 y resto, 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

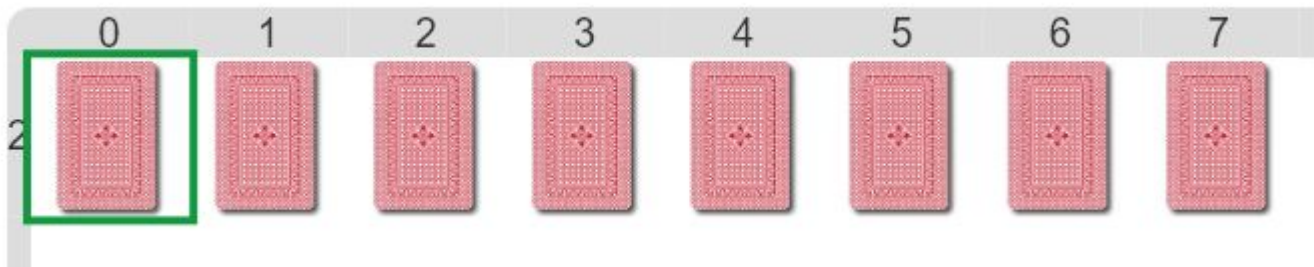
Éstas son solo algunas de las que podemos hacer

mínimo elemento

último elemento

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

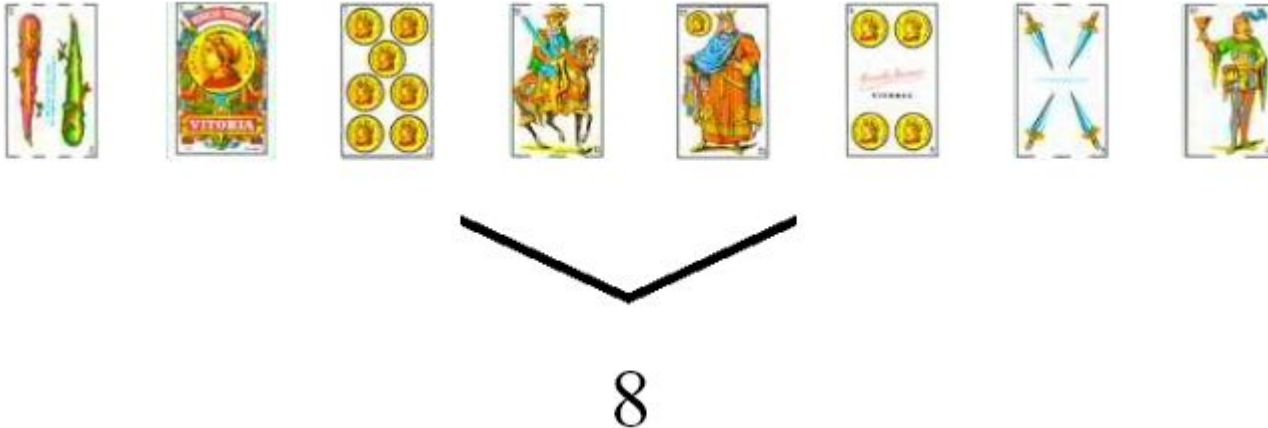
```
IniciarRecorrido()  
while (quedanElementos()) {  
    ProcesarElemento()  
    PasarAlSiguienteElemento()  
}  
FinalizarRecorrido()
```



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!



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 cantidadDeElementos(lista) {  
  /* PROPÓSITO: describe la cantidad de elementos de la lista  
  PRECONDICIÓN: ninguna  
  PARÁMETROS: lista es de tipo Lista de cualquier tipo  
  RESULTADO: un valor de tipo Número  
  OBSERVACIÓN: es un recorrido sobre la lista  
  */  
  ...  
}
```



¡PRIMERO EL CONTRATO!



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

```
function cantidadDeElementos(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)  
          

```

```
        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 cantidadDeElementos(lista) {  
  /*  */  
  listaDeLosQueFaltan := lista    // Al principio faltan todos  
  cantidadContada := 0           // y no conté ninguno aún  
  while quedanElementos (la lista de los que faltan no está vacía)  
    ProcesarElementoActual (contar el primero)  
    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 cantidadDeElementos(lista) {  
  /*  */  
  listaDeLosQueFaltan := lista    // Al principio faltan todos  
  cantidadContada := 0           // y no conté ninguno aún  
  while (not esVacía(listaDeLosQueFaltan)) {  
    ProcesarElementoActual (contar el primero)  
    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 cantidadDeElementos(lista) {  
  /*  */  
  listaDeLosQueFaltan := lista    // Al principio faltan todos  
  cantidadContada := 0           // y no conté ninguno aún  
  while (not esVacía(listaDeLosQueFaltan)) {  
    cantidadContada := cantidadContada + 1 // Cuenta el 1ro  
    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 cantidadDeElementos(lista) {  
    /*  */  
    listaDeLosQueFaltan := lista    // Al principio faltan todos  
    cantidadContada := 0           // y no conté ninguno aún  
    while (not esVacía(listaDeLosQueFaltan)) {  
        cantidadContada := cantidadContada + 1 // Cuenta el 1ro  
        listaDeLosQueFaltan := sinElPrimero(listaDeLosQueFaltan)  
        // Saca el primero de entre los que faltan contar  
    }  
    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 cantidadDeElementos(lista) {  
    /*  */  
    listaDeLosQueFaltan := lista    // Al principio faltan todos  
    cantidadContada := 0           // y no conté ninguno aún  
    while (not esVacía(listaDeLosQueFaltan)) {  
        cantidadContada := cantidadContada + 1 // Cuenta el 1ro  
        listaDeLosQueFaltan := sinElPrimero(listaDeLosQueFaltan)  
        // Saca el primero de entre los que faltan contar  
    }  
    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 mínimoElemento(lista) {  
  /* PROPÓSITO: describe el elemento más chico de la lista  
  PRECONDICIÓN: la lista no es vacía  
  PARÁMETROS: lista es de tipo Lista de cualquier tipo básico  
  RESULTADO: un valor del tipo de los elementos de la lista  
  OBSERVACIÓN: es un recorrido sobre la lista  
  */  
  ...  
}
```

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 mínimoElemento(lista) {
```

```
  /*  */
```

```
  IniciarProcesamiento (recordar que el mínimo visto es el primero)
```

```
  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 mínimoElemento(lista) {  
  /*  */  
  mínimoVisto := primero(lista)      // El 1ro es el mínimo hasta 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 mínimoElemento(lista) {  
  /*  */  
  mínimoVisto := primero(lista)           // El 1ro es el mínimo hasta ahora  
  listaDeLosQueFaltan := sinElPrimero(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 mínimoElemento(lista) {  
    /*  */  
    mínimoVisto := primero(lista)           // El 1ro es el mínimo hasta ahora  
    listaDeLosQueFaltan := sinElPrimero(lista) // y faltan mirar los demás  
    while (not esVacía(listaDeLosQueFaltan)) {  
        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 mínimoElemento(lista) {  
  /*  */  
  mínimoVisto := primero(lista)           // El 1ro es el mínimo hasta ahora  
  listaDeLosQueFaltan := sinElPrimero(lista) // y faltan mirar los demás  
  while (not esVacía(listaDeLosQueFaltan)) {  
    mínimoVisto := mínimoEntre(primero(listaDeLosQueFaltan), mínimoVisto)  
    // El primero que falta podría ser más chico...  
    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 mínimoElemento(lista) {  
  /*  */  
  mínimoVisto := primero(lista)      // El 1ro es el mínimo hasta ahora  
  listaDeLosQueFaltan := sinElPrimero(lista) // y faltan mirar los demás  
  while (not esVacía(listaDeLosQueFaltan)) {  
    mínimoVisto := mínimoEntre(primero(listaDeLosQueFaltan), mínimoVisto)  
    // El primero que falta podría ser más chico...  
    listaDeLosQueFaltan := sinElPrimero(listaDeLosQueFaltan)  
    // Saca el primero de entre los que faltan mirar  
  }  
  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 mínimoElemento(lista) {  
    /*  */  
    mínimoVisto := primero(lista)           // El 1ro es el mínimo hasta ahora  
    listaDeLosQueFaltan := sinElPrimero(lista) // y faltan mirar los demás  
    while (not esVacía(listaDeLosQueFaltan)) {  
        mínimoVisto := mínimoEntre(primero(listaDeLosQueFaltan), mínimoVisto)  
        // El primero que falta podría ser más chico...  
        listaDeLosQueFaltan := sinElPrimero(listaDeLosQueFaltan)  
        // Saca el primero de entre los que faltan mirar  
    }  
    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(elemento1, elemento2) {  
  /* PROPÓSITO: describe el mínimo entre 2 elementos  
  PRECONDICIÓN: ninguna  
  PARÁMETROS: dos elementos del mismo tipo básico  
  RESULTADO: un valor del mismo tipo de la entrada  
  OBSERVACIÓN: si son iguales, da cualquiera de ellos  
  */  
  return (choose elemento1 when (elemento1 < elemento2)  
          elemento2 otherwise)  
}
```



- 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 númerosDeLasCartas(mazo) {  
  /* PROPÓSITO: describe la lista de números de  
    las cartas del mazo  
    PRECONDICIÓN: ninguna  
    PARÁMETROS: mazo es de tipo Lista de Cartas  
    RESULTADO: un valor de tipo Lista de Números  
    OBSERVACIÓN: es un recorrido sobre la lista  
  */  
  ...  
}
```

¡¡NO OLVIDAR!!
PRIMERO EL CONTRATO

En este caso, hay que
armar una lista
resultado



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

Es un recorrido de
listas

```
function númerosDeLasCartas(lista) {  
  /*  */
```

```
    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 númerosDeLasCartas(lista) {  
    /*  */  
    listaDeLosQueFaltan := lista    // Al principio faltan todos  
    IniciarAcumulación (recordar que no transformé ninguno)  
    while (not esVacía(listaDeLosQueFaltan)) {  
        ProcesarElementoActual (transformar el primero)  
        listaDeLosQueFaltan := sinElPrimero(listaDeLosQueFaltan)  
        // Saca el primero de entre los que falta procesar  
    }  
    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

El acumulador es una lista

```
function númerosDeLasCartas(lista) {  
  /*  */
```

IniciarRecorrido (recordar que faltan todos)

```
  listaDeNúmerosVistos := [] // y no tengo ningún número
```

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

```
  listaDeNúmerosVistos := listaDeNúmerosVistos  
    ++ [ número(primeros(listaDeLosQueFaltan)) ]
```

PasarAlSiguienteElemento (recordar que saqué el primero)

```
}
```


FinalizarRecorrido (describir el resultado final)

```
}
```



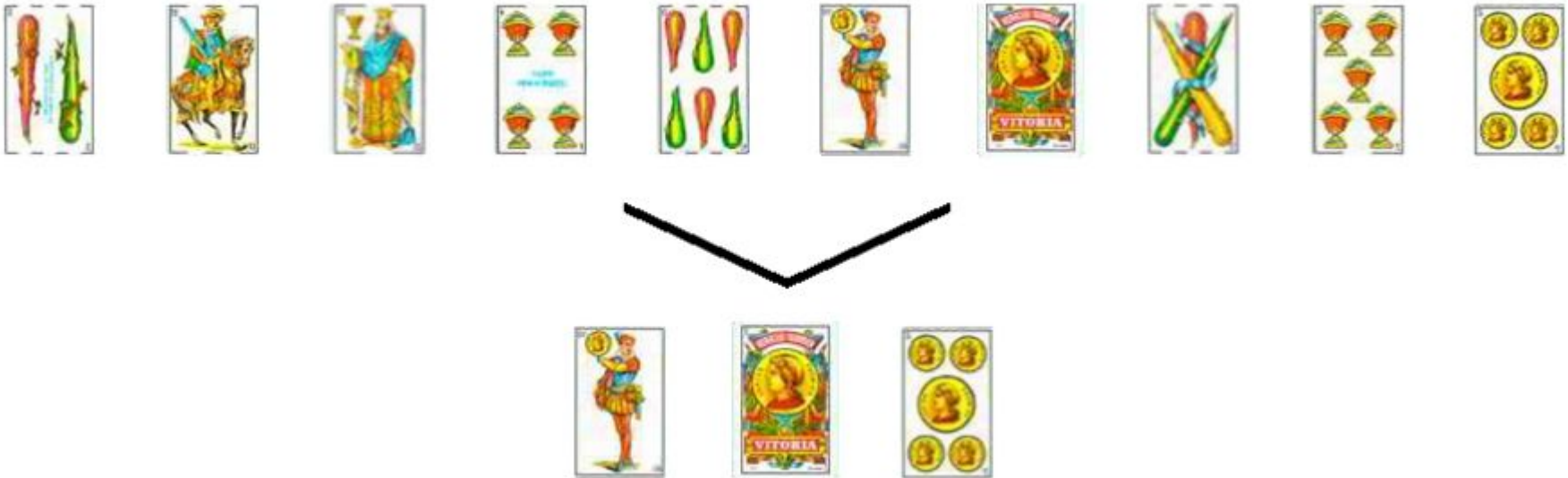

- Por ejemplo, transformar una lista
 - Se recorre, y en cada paso, se transforma el actual

Procesar es agregar el número de la 1er carta

```
function númerosDeLasCartas(lista) {  
  /*  */  
  listaDeLosQueFaltan := lista    // Al principio faltan todos  
  listaDeNúmerosVistos := []      // y no tengo ningún número  
  while (not esVacía(listaDeLosQueFaltan)) {  
    listaDeNúmerosVistos := listaDeNúmerosVistos  
                          ++ [ número(primerO(listaDeLosQueFaltan)) ]  
    listaDeLosQueFaltan := sinElPrimerO(listaDeLosQueFaltan)  
    // Saca el primero de entre los que falta procesar  
  }  
  return (listaDeNúmerosVistos)  
}
```



- 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 soloLosOros(mazo) {  
  /* PROPÓSITO: describe la lista de cartas  
    de Oros del mazo dado  
    PRECONDICIÓN: ninguna  
    PARÁMETROS: mazo es de tipo Lista de Cartas  
    RESULTADO: un valor de tipo Lista de Cartas  
    OBSERVACIÓN: es un recorrido sobre la 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 soloLosOros(mazo)
/*  */
listaDeLosQueFaltan := mazo // Al principio faltan todos
listaDeOrosVistos := []      // y no ví ninguna carta
while (not esVacía(listaDeLosQueFaltan)) {
    // Agrego la carta, solo si es de oros
    listaDeOrosVistos := listaDeOrosVistos
    ++ singularSi(primerO(listaDeLosQueFaltan)
                  , esDeOros(primerO(listaDeLosQueFaltan)))
    listaDeLosQueFaltan := sinElPrimerO(listaDeLosQueFaltan)
    // Saca el primero de entre los que falta procesar
}
return (listaDeOrosVistos)
}
```

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 singularSi(elemento, condición) {  
  /* PROPÓSITO: describe una lista, ya sea singular o vacía  
    según la condición dada  
  PRECONDICIÓN: ninguna  
  PARÁMETROS: elemento es de un tipo cualquiera  
    condición es de tipo Booleano  
  RESULTADO: un valor de tipo Lista del tipo del elemento dado  
  */  
  return (choose [elemento] when (condición)  
    [] otherwise)  
}
```

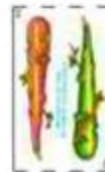



- Por ejemplo, buscar un elemento
 - Se recorre, y si encuentra lo buscado, termina (y si no, termina cuando no hay más elementos)



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 estáElAnchoDeEspadas(mazo) {  
  /* PROPÓSITO: indica si el mazo dado contiene  
    al ancho de Espadas  
  PRECONDICIÓN: ninguna  
  PARÁMETROS: mazo es de tipo Lista de Cartas  
  RESULTADO: un valor de tipo Booleano  
  OBSERVACIÓN: es un recorrido de búsqueda  
  */  
  ...  
}
```

¡Sí! ¡EL CONTRATO!



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

```
function estáElAnchoDeEspadas(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 estáElAnchoDeEspadas(mazo) {  
    /*  */  
    listaDeLosQueFaltan := mazo  
    while  
        && quedanElementos (la lista de los que faltan no está vacía)  
        y no encontréLoBuscado (la primera no es el ancho)  
        listaDeLosQueFaltan := sinElPrimero(listaDeLosQueFaltan)  
}
```

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 estáElAnchoDeEspadas(mazo) {  
    /*  */
```

La condición de fin
requiere circuito corto

IniciarRecorrido (recordar que faltan todos)

```
while (not esVacía(listaDeLosQueFaltan)  
    && not esAnchoDeEspadas(primerO(listaDeLosQueFaltan))  
    ) {
```


PasarAlSiguienteElemento (recordar que saqué el primero)

FinalizarRecorrido (describir el resultado final)

¿Cómo saber si encontré lo buscado?



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

```
function estáElAnchoDeEspadas(mazo) {  
    /**/  
    listaDeLosQueFaltan := mazo  
    while (not esVacía(listaDeLosQueFaltan)  
        && not esAnchoDeEspadas(primerO(listaDeLosQueFaltan))  
        ) {  
        listaDeLosQueFaltan := sinElPrimero(listaDeLosQueFaltan)  
    }  
    return (not esVacía(listaDeLosQueFaltan))  
    // Si quedan cartas, es porque encontré lo buscado  
}
```

Si encontró 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



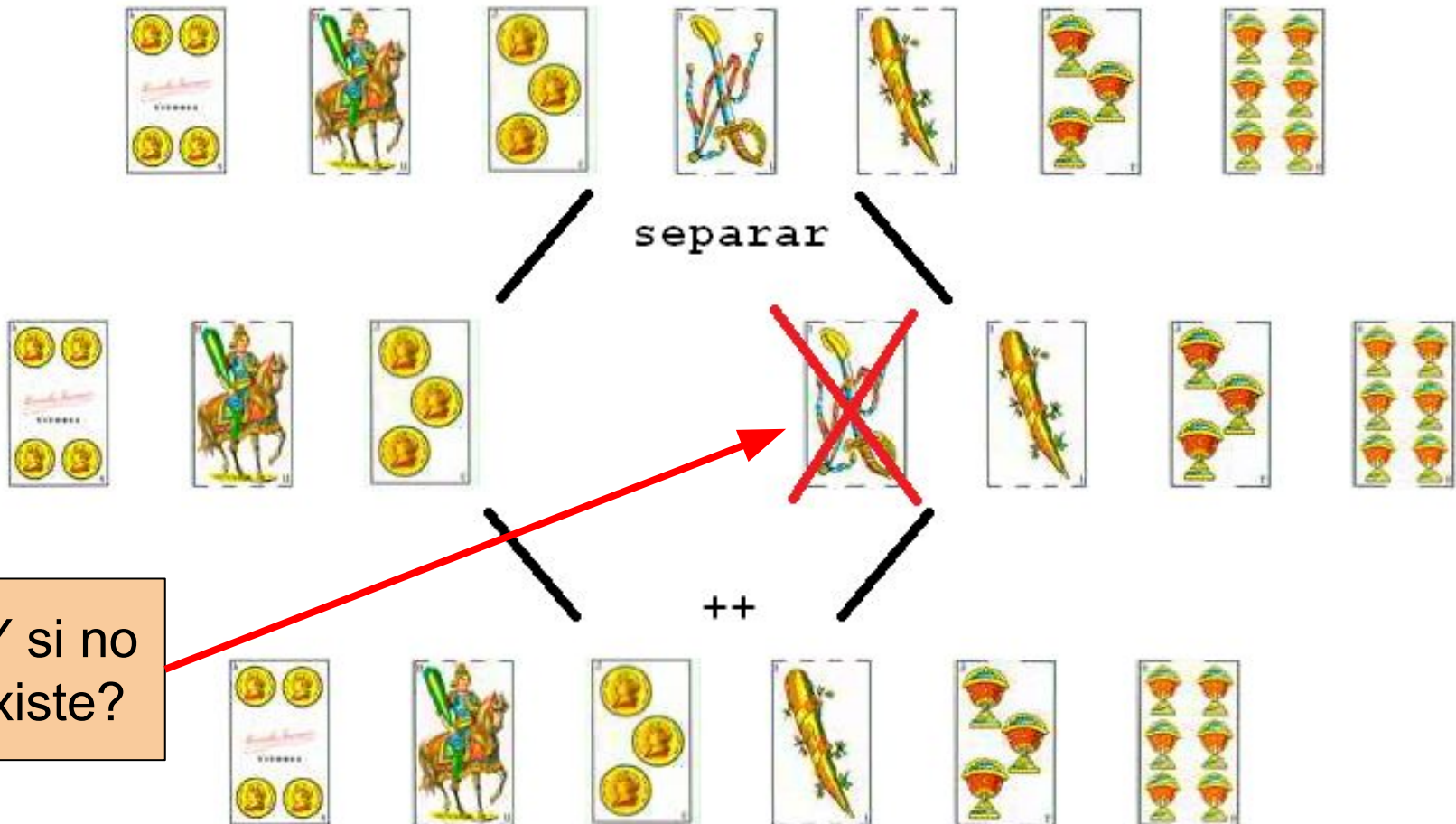
- ¿Cómo hacer si quiero eliminar una carta?
 - Hay una sola que cumple la condición

¿Cómo hacerla sin
recorrer TODA la lista?

```
function sinElAnchoDeEspadas(mazo) {  
  /* PROPÓSITO: describe al mazo, pero sin el ancho de Espadas  
  PRECONDICIÓN: ninguna  
  PARÁMETROS: mazo es de tipo Lista de Cartas  
  RESULTADO: un valor de tipo Lista de Cartas  
  OBSERVACIÓN: usa la idea de separar la lista en 2  
  */  
  ...  
}
```




- ¿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: describe al mazo, pero sin el ancho de Espadas  
  PRECONDICIÓN: ninguna  
  PARÁMETROS: mazo es de tipo Lista de Cartas  
  RESULTADO: un valor de tipo Lista de Cartas  
  OBSERVACIÓN: usa la idea de separar la lista en 2  
  */  
  return (choose cartasAntesDelAnchoDeEspadas(mazo)  
    ++ sinElPrimero(cartasDesdeElAnchoDeEspadas(mazo))  
      when (estáElAnchoDeEspadas(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 cartasAntesDelAnchoDeEspadas(mazo) {  
  /* PROPÓSITO: describe la lista con las cartas del mazo  
    que están antes del ancho de Espadas  
  PRECONDICIÓN: ninguna  
  PARÁMETROS: mazo es de tipo Lista de Cartas  
  RESULTADO: un valor de tipo Lista de Cartas  
  OBSERVACIONES:  
    * es un recorrido de búsqueda con acumulación  
  */  
  ...  
}  
  
function cartasDesdeElAnchoDeEspadas(mazo) {  
  /* PROPÓSITO: describe la lista con las cartas del mazo  
    que están desde el ancho de Espadas en adelante  
  PRECONDICIÓN: ninguna  
  PARÁMETROS: mazo es de tipo Lista de Cartas  
  RESULTADO: un valor de tipo Lista de Cartas  
  OBSERVACIONES: es un recorrido de búsqueda  
  */  
  ...  
}
```

¿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 cartasDesdeElAnchoDeEspadas(mazo) {  
    /*  */  
    listaDeLosQueFaltan := mazo  
    while (not esVacía(listaDeLosQueFaltan)  
        && not esAnchoDeEspadas(primeros(listaDeLosQueFaltan))  
        ) {  
        listaDeLosQueFaltan := sinElPrimero(listaDeLosQueFaltan)  
    }  
    return (listaDeLosQueFaltan)  
}
```

Un recorrido de búsqueda
que retorna una lista

Primero describimos
la “mitad” de atrás



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

```
function cartasAntesDelAnchoDeEspadas(mazo) {  
  /**/  
  listaDeLosQueFaltan := mazo  
  listaDeCartasVistas := []  
  while (not esVacía(listaDeLosQueFaltan)  
    && not esAnchoDeEspadas(primero(listaDeLosQueFaltan))  
    ) {  
    listaDeCartasVistas := listaDeCartasVistas  
                        ++ [ primero(listaDeLosQueFaltan) ]  
    listaDeLosQueFaltan := sinElPrimero(listaDeLosQueFaltan)  
  }  
  return (listaDeCartasVistas)  
}
```

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 estáElAnchoDeEspadas_Modular(mazo) {  
    /* PROPÓSITO: indica si el mazo dado contiene  
       al ancho de espadas, y falso en otro caso  
    PRECONDICIÓN: ninguna  
    PARÁMETROS: mazo es de tipo Lista de Cartas  
    RESULTADO: un valor de tipo Booleano  
    OBSERVACIÓN: mira la lista desde el ancho para saber  
    */  
    return (not esVacía(cartasDesdeElAncho(mazo)))  
}
```

La operación de búsqueda la hace la subtarea



- 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: describe la suma de todos los precios dados  
  PRECONDICIÓN: ninguna  
  PARÁMETROS: carritoDeCompras es de tipo Lista de Productos  
  RESULTADO: un valor de tipo Número  
  OBSERVACIÓN: es un recorrido 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  
  productosQueFaltan := carritoDeCompras  
  while (not esVacía(productosQueFaltan)) {  
    montoHastaAhora := montoHastaAhora  
                        + precio(primeros(productosQueFaltan))  
    productosQueFaltan := sinElPrimero(productosQueFaltan)  
  }  
  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_(carritoDeCompra)
/*  */
montoHastaAhora := 0
  IniciarRecorrido (recordar que faltan todos)
  while quedanElementos (la lista de los que faltan no está vacía)
    montoHastaAhora := montoHastaAhora
                      + precio(primeros(productosQueFaltan))
    PasarAlSiguienteElemento (recordar que saqué el primero)
  }
return (montoHastaAhora)
}
```

¡Estas operaciones son siempre iguales!

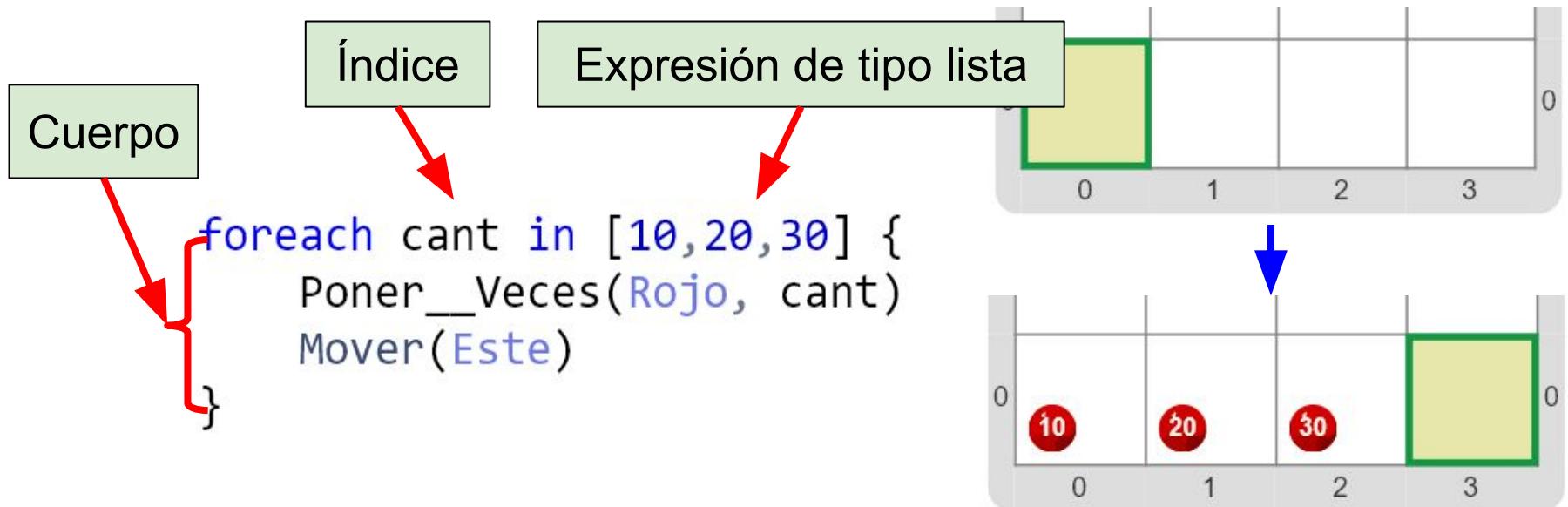


- Los recorridos se pueden expresar con una operación primitiva (siempre que NO sean de búsqueda)
 - Se llama **repetición indexada**
 - La palabra clave es **foreach** (para cada uno) y podemos leerla como “recorrer cada”

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

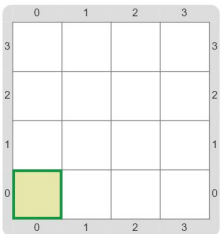
El índice toma el valor del elemento actual en cada repetición

- 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



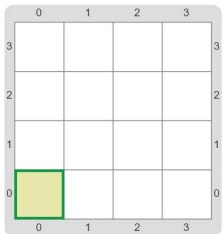
```
Poner(Verde)
foreach dir in (con_veces_(3,Norte)
               ++con_veces_(2,Este)++[Sur]) {
    Mover(dir)
    Poner(Verde)
}
```



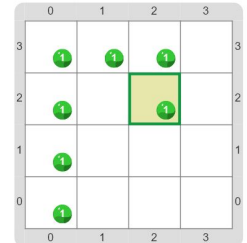
```
function con_veces_(cantidad,elemento) {
    /*...*/
    listaArmadaHastaAhora := []
    repeat(cantidad) {
        listaArmadaHastaAhora :=
            listaArmadaHastaAhora ++ [elemento]
    }
    return(listaArmadaHastaAhora)
}
```




- 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

- ***Procesamiento de listas***

- Para procesar listas usamos la estructura de recorrido
 - Usando **primero**, **esVacía** y **sinElPrimero**
- 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)