Recorrido de listas - esquema de distintos casos

La idea de este documento es resumir algunos casos clásicos de recorrido de listas.
Para cada caso vamos a dar dos ejemplos, uno con una lista de números, y el otro con una estructura más compleja que incluye varias listas. Esto lo hacemos para que se vea que las dos funciones que quedan armadas son **muy** similares.

Estructura que vamos a usar para los ejemplos

Esta representación de un distrito electoral.

```
type Distrito is record {
  field mesas           -- [número de mesa]
  field electores       -- [número de DNI de un elector]
  field padron          -- [Asignacion]
  field votaron         -- [número de DNI de un elector]
}

type Asignacion is record {
  field mesa            -- número de mesa
  field elector         -- número de DNI de un elector
}
```

Supongamos que bernal es un distrito.
En la lista `electores(bernal)` aparecen los DNI de todos los electores de Bernal. En la lista `votaron(bernal)` aparecen los DNI de los electores de Bernal que ya votaron. Obviamente, si los pienso como conjuntos, votaron es subconjunto de electores.

Cada `Asignacion` indica en qué mesa le corresponde votar a cada elector. La lista `padron(bernal)` es la lista de asignaciones del distrito: en qué mesa debe votar cada elector.

Búsqueda

En qué casos sirve

Cuando queremos obtener un elemento de una lista que cumpla una condición.
Un caso común es que tengo una lista de registros con un "código" para cada uno (p.ej. para una lista de personas, el código podría ser el DNI) y quiero obtener el registro que corresponde a ese código.

Cómo es la estrategia - ¿Va foreach o va while?
Recorro la lista hasta que encuentro lo que necesito, en el ejemplo, hasta encontrar una persona con el DNI que me pasaron. Pensar en que harías si tuvieras que buscar en un fichero, es lo mismo, vas revisando las fichas, cuando encontrás la que buscabas, parás.
Es más natural con while, poque si encuentro un elemento que cumpla, entonces puedo cortar ahí el recorrido, no necesito ir hasta el final de la lista.

La condición del while es

```
   while (not ...head(rec) cumple con la condición...) {
```

si la condición es compleja, la encerramos entre paréntesis y le ponemos el not afuera. P.ej. si queremos la primer asignación del padrón que vote en la mesa 1 o en la mesa 2, es

```
   while (not (mesa(head(rec)) == 1 || mesa(head(rec)) == 2)) {
```

Otra opción es una función auxiliar

```
   while (not votoEnMesaUnoODos(head(rec))) {
...

function votoEnMesaUnoODos(asignacion) {
  return (mesa(asignacion) == 1 || mesa(asignacion) == 2)
}
```

Ejemplo 1 - con números
Quiero obtener el primer par de una lista de números. Precondición: la lista tiene al menos un número par.

```
function primerPar(listaDeNumeros) {
  rec := listaDeNumeros
  while (not esPar(head(rec))) {
    rec := tail(rec)
  }
  return (head(rec))
}

function esPar(numero) {
  return (numero mod 2 == 0)
}
```

Ejemplo 2 - con estructuras más complejas

Quiero obtener a qué mesa está asignada la persona con un determinado DNI. Precondición: la persona está asignada a una mesa.

```
function mesaAsignada(dni,distrito) {
  rec := padron(distrito)
  while (not elector(head(rec)) == dni) {
    rec := tail(rec)
  }
  return (mesa(head(rec)))
}
```

En este ejemplo se ve que se puede devolver, en lugar de directamente `head(rec)`, una función (en este caso un proyector) aplicado a `head(rec)`.

Para darse cuenta de qué es lo que hay que devolver, puede servir en el tipo. En este caso, `mesaEnLaQueVoto` debe devolver un número de mesa. Si devolvemos `head(rec)`, entonces estaríamos devolviendo una Asignacion. Ahí te das cuenta que está mal, que tiene que ser `mesa(head(rec))`.

¿Hay alguno que …?

En qué casos sirve
Cuando queremos saber si en una lista hay algún elemento que cumpla una condición o no. P.ej. queremos saber si en una lista de personas hay alguna con el DNI tal.

Cómo es la estrategia - ¿Va foreach o va while?
Busco hasta encontrar alguno que cumpla la condición, o hasta que se termine la lista.
Es una variante de búsqueda, por eso puede andar while.
¿Como me doy cuenta que se terminó la lista? Porque rec quedó vacía.
¿Qué devolver?

- Si rec tiene algo, quiere decir que head(rec) cumple la condición, en el ejemplo es una persona con el DNI que me pasaron. Entonces devuelvo True.
- Si llegué hasta el final, quiere decir que no encontré lo que buscaba, hay que devolver False.

Ejemplo 1 - con números
Quiero saber si una lista de números tiene un número par o no.

```
function hayAlgunPar(listaDeNumeros) {
  rec := listaDeNumeros
  while (tieneAlgo(rec) && not esPar(head(rec))) {
    rec := tail(rec)
  }
  return (tieneAlgo(rec))
}
```

Ejemplo 2 - con estructuras más complejas
Quiero saber si un determinado DNI está asignado o no a una mesa.

```
function tieneMesaAsignada(dni,distrito) {
  rec := padron(distrito)
  while (tieneAlgo(rec) && not elector(head(rec)) == dni) {
    rec := tail(rec)
  }
  return (tieneAlgo(rec))
}
```

¿Será cierto que todos …?

En qué casos sirve
Cuando queremos saber si en una lista es cierto que todos los elementos cumplen o no una condición. P.ej. si queremos saber si en una lista de personas es cierto que el DNI de todas ellas es menor a 40 millones.

Cómo es la estrategia - ¿Va foreach o va while?
Es parecido al anterior, por eso puede andar while. Pero ahora busco hasta encontrar alguno que **no** cumpla la condición, o hasta que se termine la lista.
¿Qué devolver?

- Si rec tiene algo, quiere decir que head(rec) no cumple la condición, entonces no es verdadero que todos la cumplen. En el ejemplo, encontré una persona con DNI mayor a 40 millones. Hay que devolver False.
- Si llegué hasta el final, quiere decir que todos cumplen la condición, hay que devolver True

Acá pensar en las leyes lógicas de primer orden..

Ejemplo 1 - con números
Quiero saber si en una lista de números son todos pares.

```
function sonTodosPares(listaDeNumeros) {
  rec := listaDeNumeros
  while (tieneAlgo(rec) && esPar(head(rec))) {
    rec := tail(rec)
  }
  return (isEmpty(rec))
}
```

Ejemplo 2 - con estructuras más complejas
Quiero saber si para todas las personas asignadas a una mesa, el número de mesa tiene un solo dígito, o sea es menor a 10.

```
function todosEnMesasDeUnDigito(distrito) {
  rec := padron(distrito)
  while (tieneAlgo(rec) && esMesaDeUnDigito(head(rec))) {
    rec := tail(rec)
  }
  return (isEmpty(rec))
}

function esMesaDeUnDigito(asignacion) {
  return (mesa(asignacion) < 10)
}
```

Filtro

En qué casos sirve
Cuando queremos obtener, a partir de una lista, los elementos que cumplen una condición.

Cómo es la estrategia - ¿Va foreach o va while?

1. Creamos una lista nueva, arranca como la lista vacía.
2. Recorremos todos los elementos, para cada uno preguntamos si cumple o no la condición, si la cumple agregamos el elemento a la lista, si no no.
3. Devolvemos la lista

Puede ir foreach, porque hay que recorrer todos los elementos de la lista.

Ejemplo 1 - con números
Para obtener los números pares de una lista de números

```
function pares(listaDeNumeros) {
  losPares := []
  foreach num in listaDeNumeros {
    if (esPar(num)) {
      losPares := losPares ++ [num]
    }
  }
  return (losPares)
}

function esPar(numero) ...  -- está hecha al hablar de búsqueda.
```

Ejemplo 2 - con estructuras más complejas
Para obtener los DNI de las personas que votaron en una mesa.

```
function votaronEnMesa(distrito,mesa) {
  losQueVotaron := []
  foreach votante in votaron(distrito) {
    if (mesaAsignada(votante,distrito) == mesa) {
      losQueVotaron := losQueVotaron ++ [votante]
    }
  }
  return (losQueVotaron)
}
```

La función `mesaAsignada(dni,distrito)` ya la desarrollamos, en la parte de búsqueda.

Recolección

En qué casos sirve
Cuando queremos obtener, a partir de una lista, el resultado de aplicar una función a cada elemento. P.ej. en una lista de personas, quiero la edad de cada una.

Cómo es la estrategia - ¿Va foreach o va while?

1. Creamos una lista nueva, arranca como la lista vacía.
2. Recorremos todos los elementos, para cada uno agregamos lo que nos piden.
3. Devolvemos la lista

Puede ir foreach, porque hay que recorrer todos los elementos de la lista.
Fíjense que es parecido a un filtro, con estas diferencias

- no hay condición.
- en vez de agregar el elemento, se agrega algo que se calcula a partir del elemento.

Ejemplo 1 - con números
Dada una lista de números, queremos obtener la lista del triple de cada número.

```
function elTripleDeCada(listaDeNumeros) {
  triples := []
  foreach num in listaDeNumeros {
    triples := triples ++ [num * 3]
  }
  return (triples)
}
```

Ejemplo 2 - con estructuras más complejas
Para obtener la lista con la cantidad de electores asignados a cada mesa, una lista de números.

```
function cantidadDeElectoresPorMesa(distrito) {
  cantidades := []
  foreach mesa in mesas(distrito) {
    cantidades := cantidades ++
                  [longitud(electoresEnMesa(mesa,distrito))]
  }
  return (cantidades)
}

-- aca tenemos otro ejemplo de filtro
function electoresEnMesa(queMesa,distrito) {
  electores := []
  foreach asignacion in padron(distrito) {
    if (mesa(asignacion) == queMesa) {
      electores := electores ++ [elector(asignacion)]
    }
  }
  return (electores)
}
```

Recolección con filtro

En qué casos sirve
Cuando queremos obtener, a partir de una lista, el resultado de aplicar una función a algunos elementos. P.ej. en una lista de personas, quiero la edad de los varones.

Cómo es la estrategia - ¿Va foreach o va while?
Es una combinación de filtro con recolección. Es la estrategia de filtro, pero cuando agregamos, en lugar de agregar el elemento, agregamos el resultado de hacer la cuenta que nos pidan.

Ejemplo 1 - con números
Dada una lista de números, queremos obtener la lista del triple de cada número par en la lista.

```
function elTripleDeCadaPar(listaDeNumeros) {
  triples := []
  foreach num in listaDeNumeros {
    if (esPar(num)) {
      triples := triples ++ [num * 3]
    }
  }
  return (triples)
}
```

Ejemplo 2 - con estructuras más complejas
Para obtener la lista con la cantidad de electores asignados a cada mesa, considerando solamente las mesas sin abrir, o sea, las mesas en las que todavía no votó nadie.

```
function cantidadDeElectoresPorMesaSinAbrir(distrito) {
  cantidades := []
  foreach mesa in mesas(distrito) {
    if (isEmpty(votaronEnMesa(distrito,mesa))) {
      cantidades := cantidades ++
                    [longitud(electoresEnMesa(mesa,distrito))]
    }
  }
  return (cantidades)
}
```

Recordar que `votaronEnMesa` está desarrollada en la parte de filtro.

Totalización

En qué casos sirve
Cuando queremos obtener, a partir de una lista, el total o la suma de un valor asociado a cada elemento.

Cómo es la estrategia - ¿Va foreach o va while?

1. Ponemos una variable en 0.
2. Recorremos todos los elementos, para cada uno sumamos lo que nos piden.
3. Devolvemos la variable.

Puede ir foreach, porque hay que recorrer todos los elementos de la lista.

Ejemplo 1 - con números
Dada una lista de números, queremos obtener la suma.

```
function sumatoria(listaDeNumeros) {
  suma := 0
  foreach num in listaDeNumeros {
    suma := suma + num
  }
  return (suma)
}
```

Ejemplo 2 - con estructuras más complejas
Para obtener la cantidad total de electores asignados a mesas en las que no votó nadie.

```
function totalElectoresEnMesasNoAbiertas(distrito) {
  total := 0
  foreach mesa in mesasSinAbrir(distrito) {
    total := total + longitud(electoresEnMesa(mesa,distrito))
  }
  return (total)
}

// ooootro filtro
function mesasSinAbrir(distrito) {
  sinAbrir := []
  foreach mesa in mesas(distrito) {
    if (esMesaSinAbrir(mesa,distrito)) {
      sinAbrir := sinAbrir ++ [mesa]
    }
  }
  return (sinAbrir)
}

function esMesaSinAbrir(mesa,distrito) {
  return (isEmtpy(votaronEnMesa(distrito,mesa)))
}
```

Nota: esto también se podría hacer como una totalización con filtro, así:

```
function totalElectoresEnMesasSinAbrir(distrito) {
```

```
  total := 0
  foreach mesa in mesas(distrito) {
    if (esMesaSinAbrir(mesa,distrito)) {
      total := total + longitud(electoresEnMesa(mesa,distrito))
    }
  }
  return (total)
}
```

Ejercicios

Tomemos este modelo de un casino.

```
type Apuesta is record {
  field apostador              -- DNI de apostador
  field nroApostado            -- número en el paño
  field monto                  -- número, cantidad de plata
}

type Mesa is record {
  field numeroMesa             -- el número de la mesa
  field apuestas               -- [Apuesta]
  field croupier               -- nro de legajo
}

type Casino is record {
  field mesas                  -- [Mesa]
  field empleados              -- [nro de legajo]
}
```

Desarrollar estas funciones, indicando a qué caso corresponde cada una

1. totalApostadoEnMesa(queMesa): devuelve el importe total apostado en la mesa.
2. primerNumeroEntre(listaDeNumeros,min,max): devuelve el primer número en la lista que esté en el rango entre min y max.
   P.ej. primerNumeroEntre([93,4,23,81,704,21],10,50) denota 23.
3. multiplosDeCinco(listaDeNumeros): eso. Recordar que un número es múltiplo de 5 si termina en 5 o en 0. La última cifra es … el resto de dividir por 10.
4. esCasinoProductivo(casino): indica si para todas las mesas en el casino, el total de apuestas por mesa es de al menos 1000 pesos.
5. numerosDeApuestasAltas(mesa): denota la lista de números que registren una apuesta de al menos 30 pesos en la mesa.
6. hayAlgunoMayorA(listaDeNumeros,min): .. eso.
   P.ej. hayAlgunoMayorA([93,4,23,81,704,21],10) denota True,
   hayAlgunoMayorA([93,4,23,81,704,21],1000) denota False.
7. estanTodosEntre(listaDeNumeros,min,max): eso.
8. apuestasAl(unaMesa,unNumero), devuelve una lista con las apuestas al número indicado en la mesa.
9. rangoDeCadaUno(listaDeNumeros): una lista con el rango de cada número en la lista, donde el rango de un número entre 0 y 9 es 0, el rango de un número entre 10 y 99 es 1, y el rango de un número mayor a 100 es 2.
   P.ej. rangoDeCadaUno([93,4,23,81,704,21]) denota [1,0,1,1,2,1]
10. rangoDeLosPares(listaDeNumeros): el rango de cada uno, pero considerando solamente los pares.
    P.ej. rangoDeLosPares([93,4,23,81,704,21]) denota [0,2] porque los únicos pares son 4 y 704.
11. apostoEn(mesa,unApostador): indica si hay al menos una apuesta del apostador indicado en la mesa.
12. primerApuestaDeMasDe(mesa,montoMinimo), devuelve la primer apuesta en la mesa cuyo monto sea mayor al mínimo.
13. croupiers(casino): denota la lista de números de legajo de los croupiers de cada mesa.

Ahora, algunos más difíciles. Tener en cuenta esta estructura adicional

```
type ApostadorEnMesa is record {
  field apostador               -- DNI de apostador
  field mesa                    -- número de mesa
  field nrosApostados           -- [número en el paño]
  field montoTotal              -- número, cantidad de plata
}
```

Allá vamos. OJO en varios conviene usar alguno de la primera parte.
Para aspirar a un 7 o más en el parcial, te tienen que salir al menos algunos de estos.

1. montoTotalApostado(casino): eso.
2. empleadosQueNoSonCroupiers(casino): la lista de los empleados del casino que no son croupiers.
3. apostadoresFuertes(casino): lista de números de DNI de los apostadores que tienen, en total contando todas las mesas, apuestas por al menos 5000 pesos.
4. croupiersQueAtiendenA(casino,dniApostador): lista de números de legajo de los croupiers que atienden al apostador indicado, o sea, que están asignados a mesas en las que el apostador hizo al menos una apuesta.
5. totalApostadoPorMesa(casino,numero): una lista de números, con el total apostado al número indicado en cada mesa del casino, en el mismo orden de `mesas(casino)`.
6. reporteApostador(casino,dniApostador): una lista de `ApostadorEnMesa` para cada mesa en la que el apostador hizo al menos una apuesta.
7. reporteMesa(mesa): una lista de `ApostadorEnMesa` para cada apostador de la mesa.