Resumen de recorridos (segundo parcial)

Partimos del conocimiento de recorridos que tenemos hasta el primer parcial. Ya conocemos varios esquemas, y hemos recorrido diversos elementos.

Para el segundo parcial, se suma poder recorrer sobre elementos enumerativos (por ej. días de la semana, meses del año, etc.), lo cual, es idéntico a recorrer direcciones o colores, con la salvedad de que se debe primero realizar las subtareas que determinen cuál es el primer elemento, cuál es el último y como se obtiene el siguiente.

La parte más interesante tiene que ver con recorrer listas. Para estas, surge una nueva herramienta que nos permite recorrerlas cómodamente, **la repetición indexada**. Además las listas tienen algunas características que las hacen diferentes en términos de recorridos.

Por ejemplo, en las listas, no hay que procesar un elemento como caso borde ya que al recorrer con repetición indexada está garantizado haber recorrido todos los elementos, y que no falte ninguno. Esto también aplica a recorrer con repetición condicional, ya que contamos con la expresión "lista vacía" que nos permite reconocer cuando se nos acabaron los elementos sin incurrir en un error (como salirnos del tablero en el caso de las celdas o volver a considerar el primer elemento en el caso de los tipos enumerativos).

Por otro lado, al usar la repetición indexada para recorrer no es necesario plantear las subtareas correspondientes a "Ir al primer elemento", "quedan elementos por recorrer" y "Pasar al siguiente elemento" (aunque esto *sí* es necesario cuando usamos la repetición condicional, pero en principio se utilizan las primitivas de listas), lo cual simplificará los esquemas realizados.

En la materia vemos 6 esquemas de recorridos en la segunda parte de la materia:

- de transformación
- de filtro
- de acumulación
- de máximo-mínimo
- de búsqueda (sabiendo que el elemento buscado existe)
- de búsqueda (sin saber si el elemento buscado existe)

Los primeros cuatro, además, pueden considerarse recorridos de totalización, ya que en todos se está buscando un total (acumulando en una variable, que puede contener un número, un elemento o una lista).

A continuación mencionaremos los distintos esquemas. Para ello, los expresaremos como funciones, donde se recibirá por parámetro la lista cuyos elementos se deben recorrer bajo el nombre de "lista" (aunque no necesariamente es siempre una lista pasada por parámetro la que debe ser recorrida). También mencionaremos como identificar cada esquema y las subtareas a pensar.

Recorrido de transformación

La transformación en una lista implica que el resultado a generar es una nueva lista, donde para cada elemento en la lista original, hay un elemento en la lista generada que le corresponde.

```
function lista_Transformada(lista) {
    listaResultadoAlMomento := []
    foreach elemento in lista {
                    listaResultadoAlMomento  :=  listaResultadoAlMomento  ++
[elemento_Transformado(elemento)]
    }
    return (listaResultadoAlMomento)
}
```

En una transformación, está garantizado por estructura del recorrido (ya que siempre hacemos ++ con una lista singular) que la longitud de la lista transformada, va a ser la misma que la de la lista original, es decir:

- **longitudDe_(lista_Transformada(lista)) == longitudDe_(lista)**

Si la lista original está vacía, la lista transformada será vacía, por lo que el recorrido no falla en caso de una lista vacía.

La lista a transformar puede ser una lista cualquiera, digamos que es de tipo “Lista de A”, siendo “A” un tipo cualquiera. La lista que se devuelve tendrá el tipo “Lista de B”, siendo “B” el tipo de la función `elemento_Transformado` (que podría ser el mismo que “A” en algunas ocasiones).

Así, este esquema requiere de una subtarea, que realice dicha transformación. La subtarea **elemento_Transformado** es la subtarea, tal que si le doy un elemento del tipo A, describe un elemento del tipo B (O al menos un valor diferente del tipo A), es decir, transformándolo. Tener en cuenta que, a pesar de llamarlo así, la lista original no es transformada sino que lo que se hace es crear una lista nueva con elementos nuevos.

Problemas que caen en la categoría de transformaciones suelen tener enunciados de la forma:

- **Dada una lista de A, describir una lista donde para cada A, se lo transformó en un B.**

Ejemplos concretos:

- Dada una lista de números, describir la lista donde a cada número se lo elevó al cuadrado.

```
function losNúmeros_ElevadosAlCuadrado(números) {
    // PARÁMETROS:
    // * números : Lista de Número
    // TIPO: Lista de Número
    cuadradosAlMomento := []
    foreach número in números {
        cuadradosAlMomento := cuadradosAlMomento ++
            [número ^ 2] // No necesito una subtarea, alcanza con ^
    }
    return (cuadradosAlMomento)
}
```

- Dada una lista de Personas, describir una lista con los nombres de las personas.

```
function nombreDePersonas_(personas) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Lista de String
    nombresAlMomento := []
    foreach persona in personas {
        nombresAlMomento := nombresAlMomento ++
            [nombre(persona)] // Alcanza a veces con la observadora

    }
    return (nombresAlMomento)
}
```

- Dada una lista de Deportes, describir la lista que contiene la cantidad de jugadores que compiten en cada deporte.

```
function jugadoresCompitiendoEnDeportes_(deportes) {
    // PARÁMETROS:
    // * deportes : Lista de Deporte
    // TIPO: Lista de Número
    competidoresPorDeporteAlMomento := []
    foreach deporte in deportes {
        competidoresPorDeporteAlMomento := competidoresPorDeporteAlMomento ++
            [cantidadCompitiendoEnDeporte_(deporte)]
            // A veces necesito una subtarea más compleja
    }
    return (competidoresPorDeporteAlMomento)
}
```

Puede darse el caso en donde la transformación es condicional, es decir, algunos elementos se transforman, mientras que otros permanecen igual. Esto solo puede ocurrir si A y B son el mismo tipo. Un ejemplo:

- Dada una lista de Autos, describir la lista de Autos donde se pintó de rojo a los autos azules.

```
function autos_ConAzulesPintadosARojo(autos) {
    // PARÁMETROS:
    // * autos : Lista de Auto
    // TIPO: Lista de Auto
    autosPintadosAlMomento := []
    foreach auto in autos {
        autosPintadosAlMomento := autosPintadosAlMomento ++
            [auto_PintadoARojoSiEsAzul(auto)]
            // A veces necesito una subtarea más compleja
    }
    return (autosPintadosAlMomento)
}

function auto_PintadoARojoSiEsAzul(auto) {
    // PARÁMETROS:
    // * auto : Auto
    // TIPO: Auto
    return (choose
        auto_PintadoDeRojo(auto) when (color(auto) == Azul)
        auto otherwise
    )
}

function auto_PintadoDeRojo(auto) {
    // PARÁMETROS:
    // * auto : Auto
    // TIPO: Auto
    return (Auto(auto | color <- Rojo))
}
```

Recorrido de filtro

El filtrado en una lista implica que el resultado a generar es una nueva lista, donde van a estar algunos (pero potencialmente no todos) los elementos de la lista original, sin modificaciones. El esquema hace uso de **singular_Si_** como forma de agregar condicionalmente un elemento a la lista de resultados. Es decir, no siempre se agrega el elemento a la lista, depende de una condición.

```
function lista_Filtrada(lista) {
    listaResultadoAlMomento := []
    foreach elemento in lista {
        listaResultadoAlMomento := listaResultadoAlMomento ++
                        singular_Si_(elemento, debeAgregarse_(elemento))
    }
    return (listaResultadoAlMomento)
}
```

En un filtro, la longitud de la lista filtrada, va a ser la misma o menor que la de la lista original, es decir:

- **longitudDe_(lista_Filtrada(lista)) <= longitudDe_(lista)**

Es decir, nada impide que el resultado no sea vacío, incluso si la lista original no lo era. Sin embargo, si la lista original estaba vacía, también lo estará el resultado, pero el recorrido sigue funcionando bien.

La lista a filtrar puede ser una lista cualquiera, digamos que es de tipo “Lista de A”, siendo “A” un tipo cualquiera. La lista que se devuelve tendrá el tipo “Lista de A”, ya que los elementos no se transforman (más aún, los elementos en la lista resultante serán elementos que ya estaban en la lista original).

Así, este esquema requiere de una subtarea **debeAgregarse_** que dado un elemento del tipo A, describe un Booleano, que indica si el elemento debe ser agregado a la lista.

Problemas que caen en la categoría de filtros suelen tener enunciados de la forma:

- **Dada una lista de A, describir la lista de los A que cumplen X.**

Ejemplos concretos:

- Dada una lista de números, describir la lista de los números pares

```
function paresEn_(números) {
    // PARÁMETROS:
    // * números : Lista de Número
    // TIPO: Lista de Número
    paresAlMomento := []
    foreach número in números {
        paresAlMomento := paresAlMomento ++
                          singular_Si_(número, esPar_(número))
    }
    return (paresAlMomento)
}
```

- Dada una lista de Personas, describir las personas mayores de edad.

```
function mayoresDeEdadEn_(personas) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Lista de Persona
    mayoresDeEdadAlMomento := []
    foreach persona in personas {
        mayoresDeEdadAlMomento := mayoresDeEdadAlMomento ++
                           singular_Si_(persona, esMayorDeEdad_(persona))
    }
    return (mayoresDeEdadAlMomento)
}
```

- Dada una lista de Deportes, describir los deportes que sean acuáticos

```
function deportesDe_QueSonAcuáticos(deportes) {
    // PARÁMETROS:
    // * deportes : Lista de Deporte
    // TIPO: Lista de Deporte
    deportesAcuáticosAlMomento := []
    foreach deporte in deportes {
        deportesAcuáticosAlMomento := deportesAcuáticosAlMomento ++
                singular_Si_(deporte, esDeporte_DeTipo_(deporte, Acuático))
    }
    return (deportesAcuáticosAlMomento)
}
```

Recorrido de acumulación

Hablamos de acumulación en una lista cuando el resultado implica calcular un número a partir de los diversos elementos de la lista.

```
function acumuladoEn_(lista) {
    acumuladoAlMomento := valorInicial
    foreach elemento in lista {
        acumuladoAlMomento := acumuladoAlMomento ⊕ valorDe_(elemento)
```

```
    }
    return (acumuladoAlMomento)
}
```

Nuevamente, al igual que en las acumulaciones de recorridos sobre tablero, la operación puede ser una suma, una multiplicación, u otra. Por lo que hay que determinar en este caso, dos cosas, la operación, y el **valor inicial**, que suele ser el neutro de la operación a realizar.

La lista sobre la cual acumular no tiene por que ser de números, sino que puede ser de cualquier tipo, es decir "Lista de A", siendo "A" un tipo cualquiera. Lo que se devuelve en este caso es un Número.

Este esquema requiere de una subtarea **valorDe_** que dado un elemento del tipo A, describe un Número, siendo dicho número el que vamos a acumular. A veces, la subtarea es trivial e innecesaria (por ejemplo, si ya estoy recorriendo números, tal vez consiste en usar directamente el elemento) o si estoy contando la cantidad de elementos, todo elemento vale uno. En otras ocasiones puede consistir en subtareas más complejas, incluso algunas que impliquen otros recorridos. En algunos escenarios, puede ser útil la subtarea **unoSi_CeroSino**.

Problemas que caen en la categoría de acumulaciones suelen tener enunciados de la forma:

- **Dada una lista de A, describir el total de X de los elementos de dicha lista.**

Ejemplos concretos:

- Dada una lista de números, describir la suma de los mismos

```
function sumatoriaDe_(números) {
    // PARÁMETROS:
    // * números : Lista de Número
    // TIPO: Número
    sumaAlMomento := 0
    foreach número in números {
        sumaAlMomento := sumaAlMomento + número
                        // Un caso donde se usa directo el elemento
    }
    return (sumaAlMomento)
}
```

- Dada una lista de cualquier cosa, determinar la longitud de la lista

```
function longitudDe_(lista) {
    // PARÁMETROS:
    // * lista : Lista de "Cualquiera"
    // TIPO: Número
    longitudAlMomento := 0
    foreach elemento in lista {
        longitudAlMomento := longitudAlMomento + 1
                        // Un caso donde se usa siempre uno
    }
    return (longitudAlMomento)
}
```

- Dada una lista de Personas, determinar la suma de edades de todas ellas

```
function edadTotalDe_(personas) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Número
    sumaDeEdadesAlMomento := 0
    foreach persona in personas {
        sumaDeEdadesAlMomento := sumaDeEdadesAlMomento + edad(persona)
    }
    return (sumaDeEdadesAlMomento)
}
```

Recorrido de máximo-mínimo

Hablamos de máximo o mínimo en una lista cuando lo que se busca es un elemento de la lista que sea más grande o más pequeño que todo el resto, para algún criterio cualquiera de lo que significa ser grande o pequeño. Así, planteamos nuevamente la idea de "elMejorPorAhora" (el más grande, el más largo, el más chico, el más joven, etc.).

```
function elMejorEn_(lista) {
    elMejorPorAhora := primero(lista)
    foreach elemento in sinElPrimero(lista) {
        elMejorPorAhora := elMejorEntre_Y_(elemento, elMejorPorAhora)
    }
    return (elMejorPorAhora)
}
```

Notar que determinar un máximo o un mínimo implica que haya al menos un elemento en la lista (más aún, podría implicar que haya uno que sea más grande o más chico que todo el resto). Por lo tanto, este tipo de recorrido requiere asumir que la lista no está vacía.

La lista sobre la cual encontrar el máximo o mínimo puede ser de cualquier tipo, es decir "Lista de A", siendo "A" un tipo cualquiera. Lo que se devuelve en este caso es uno de los elementos, es decir, uno de esos "A".

Este esquema requiere de una subtarea **elMejorEntre_Y_** que dados dos elementos del tipo "A" (Lo que sea que haya en la lista), describe uno de ellos, aquel que cumple mejor el criterio de máximo o mínimo buscado.

Problemas que caen en la categoría de máximo o mínimo suelen tener enunciados de la forma:

- **Dada una lista de A, describir el elemento más X de la lista.**

Ejemplos concretos:

- Dada una lista de números, describir el número más grande

```
function elMásGrandeEn_(números) {
    // PARÁMETROS:
    // * números : Lista de Número
    // TIPO: Número
    elMásGrandeAlMomento := primero(números)
    foreach número in sinElPrimero(números) {
        elMásGrandeAlMomento := mayorEntre_Y_(número, elMásGrandeAlMomento)
                        // Un caso donde se usa la función de biblioteca
    }
    return (elMásGrandeAlMomento)
}
```

- Dada una lista de cualquier Personas, determinar la persona más joven

```
function másJovenEntre_(personas) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Persona
    másJovenAlMomento := primero(personas)
    foreach persona in sinElPrimero(personas) {
        másJovenAlMomento := elMásJovenEntre_Y_(persona, másJovenAlMomento)
    }
    return (másJovenAlMomento)
}
```

- Dada una lista de cualquier Guerreros, determinar el guerrero con más poder

```
function elMásPoderososEntre_(guerreros) {
    // PARÁMETROS:
    // * personas : Lista de Guerrero
    // TIPO: Guerrero
    másPoderosoAlMomento := primero(guerreros)
    foreach guerrero in sinElPrimero(guerreros) {
        másPoderosoAlMomento :=
                elMásPoderosoEntre_Y_(guerrero, másPoderosoAlMomento)
    }
    return (másPoderosoAlMomento)
}
```

Recorrido de búsqueda (sabiendo que está el elemento)

Un recorrido de búsqueda (sabiendo que el elemento está) se da cuando queremos en general obtener un elemento de la lista que cumple algún criterio particular. Este esquema no utiliza foreach, sino while, por lo que usará una variable auxiliar para recorrer la lista.

```
function recuperarElementoEn_(lista) {
    pendientesPorVer := lista
    while (not esElBuscado_(primero(pendientesPorVer))) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (primero(pendientesPorVer))
}
```

En los recorridos de búsqueda, a diferencia de los de totalización, no se requiere de una variable para acumular resultado, ya que lo que se quiere determinar es si algo está o no en la lista, o recuperarlo sí es que está allí. Tampoco se van a recorrer necesariamente todos los elementos de la lista, ya que se termina ni bien se encontró lo que se buscaba.

En el caso de las búsquedas sabiendo que el elemento está, se suele querer encontrar u obtener cuál es el elemento que cumple una condición determinada, sabiendo a priori que hay uno que la cumple. Es decir, se requiere que haya algún elemento que cumpla la condición.

La lista sobre la cual buscar puede ser de cualquier tipo, es decir “Lista de A”, siendo “A” un tipo cualquiera. Lo que se devuelve, en general, para las búsquedas en las que se sabe que el elemento está, es uno de los elementos, es decir, uno de esos “A” (el que cumple la condición, o el primero de los que la cumplen).

Este esquema requiere de una subtarea **esElBuscado_** que dado un elemento del tipo “A” (Lo que sea que haya en la lista), describe un Booleano que indica si el elemento es o no el que se está buscando.

Problemas que caen en la categoría de búsqueda sabiendo que está suelen tener la forma de:

- **Dada una lista de A, describir el elemento que cumple X.**

Ejemplos concretos:

- Dada una lista de números, y sabiendo que hay en ella solo un número que es primo, describir el número primo de la lista.

```
function primoEn_(números) {
    // PARÁMETROS:
    // * números : Lista de Número
    // TIPO: Número
    pendientesPorVer := números
    while (not esPrimo_(primero(pendientesPorVer))) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (primero(pendientesPorVer))
}
```

- Dada una lista de cualquier Personas en donde no hay dos con el mismo DNI, describir la Persona con DNI 1234

```
function personaConDNI1234En_(personas) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Persona
    pendientesPorVer := personas
    while (dni(primero(pendientesPorVer)) /= “1234”) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (primero(pendientesPorVer))
}
```

- Dada una lista de cualquier Guerreros, describir el primer guerrero que use un hacha, sabiendo que hay al menos uno en la lista.

```
function elPrimeroQueUsaHachaEn_(guerreros) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Persona
    pendientesPorVer := guerreros
    while (not usaHacha_(primero(pendientesPorVer))) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (primero(pendientesPorVer))
}
```

Recorrido de búsqueda (sin saber si está el elemento)

Un recorrido de búsqueda (sin saber si está el elemento) se da cuando queremos verificar si en la lista hay algún elemento que cumpla determinada condición (o que no la cumpla). A diferencia del anterior, funciona sin restricciones sobre la lista.

```
function existeElementoEn_(lista) {
    pendientesPorVer := lista
    while (not esVacía(pendientesPorVer)
              && not esElBuscado_(primero(pendientesPorVer))) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (not esVacía(pendientesPorVer))
}
```

Notar que la subtarea **esElBuscado_** funciona igual que para el recorrido anterior, por lo que se pueden recorrer el mismo estilo de listas, y de la misma forma, y la subtarea hasta puede ser la misma.

Lo que cambia suele ser el return, que implica ver si efectivamente encontramos el elemento buscado. Por lo que este tipo de recorridos describen un Booleano.

Problemas que caen en la categoría de búsqueda sin saber que el elemento está suelen tener la forma de:

- **Dada una lista de A, indicar sí hay un elemento que cumple X.**

Ejemplos concretos:

- Dada una lista de números, indica si hay un número primo en la lista.

```
function hayPrimoEn_(números) {
    // PARÁMETROS:
    // * números : Lista de Número
    // TIPO: Booleano
    pendientesPorVer := números
    while (not esVacía(pendientesPorVer)
              && not esPrimo_(primero(pendientesPorVer))) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (not esVacía(pendientesPorVer))
}
```

- Dada una lista de Personas, indicar sí hay una con DNI 1234

```
function hayPersonaConDNI1234En_(personas) {
    // PARÁMETROS:
    // * personas : Lista de Persona
    // TIPO: Booleano
    pendientesPorVer := personas
    while (not esVacía(pendientesPorVer)
              && dni(primero(pendientesPorVer)) /= “1234”) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (not esVacía(pendientesPorVer))
}
```

- Dada una lista de cualquier Guerreros, indicar si hay uno que use un hacha.

```
function hayGuerreroQueUsaHachaEn_(guerreros) {
    // PARÁMETROS:
    // * personas : Lista de Guerrero
    // TIPO: Booleano
    pendientesPorVer := guerreros
    while (not esVacía(pendientesPorVer)
              && not usaHacha_(primero(pendientesPorVer))) {
        pendientesPorVer := sinElPrimero(pendientesPorVer)
    }
    return (not esVacía(pendientesPorVer))
}
```

Conclusiones:

Al igual que con los recorridos sobre tablero, los esquemas son una guía útil, pero no son regla fija. Hay problemas que no se pueden resolver aplicando un esquema de los mencionados, y requieren pensar un poco más, usando partes de diferentes esquemas (por ej. una búsqueda con una acumulación, o una transformación con filtro). Adicionalmente, puede que haya problemas que calcen casi perfecto en el esquema, pero que requieran leves modificaciones, por ej. una búsqueda cuando lo que queremos es saber que el elemento NO está en la lista.

Por otro lado, vemos un criterio claro sobre cuándo usar un tipo de repetición u otra. Usaremos repetición indexada cuando tengamos que recorrer la totalidad de la lista, un elemento a la vez, y estemos recorriendo una única lista. Usaremos repetición condicional en otros escenarios.