Introducción a Listas

Introducción a la programación

Tecnicatura en Programación Informática,
Universidad Nacional de Quilmes

Introducción a la programación

¿Qué vemos hoy?

XGobstones

- XGobstones es una extensión de Gobstones que provee:
  - Listas y registros: que vemos en la materia
  - Procedimientos sobre otros tipos: no lo vemos

Motivación I

- Supongamos que estamos programando el juego Sokoban
- Para evitar tener que reiniciar el juego cada vez que se pierde, se quiere ofrecer la posibilidad de deshacer una movida.

- ¿Sugerencias para implementarlo?

Motivación II

- Estamos programando el buscaminas.
- Queremos usar el tablero y las bolitas para visualizar, pero no queremos que se puedan ver las pistas y minas debajo de las celdas tapadas

- ¿Sugerencias para implementarlo?

Idea informal

- Una lista es una colección de valores llamados elementos
  - Sokoban: lista de jugadas
  - Buscaminas: lista de minas, lista de lugares destapados
- Cada valor puede aparecer muchas veces en la lista
  - Sokoban: movimiento hacia la derecha
- Los valores tienen un orden en la lista
  - Sokoban: ¿primero arriba o a la derecha?
  - Buscaminas: posición de una mina = posición en un recorrido del tablero

Definición

Definición

Una lista es una colección (finita) ordenada de valores de un mismo tipo.

- Las listas se escriben $[v_1, v_2, \ldots, v_n]$ donde:
  1. v_1 es el primer elemento,
  2. v_2 es el segundo elemento,
  3. v_3 es el tercer elemento,
  4. ...
  5. v_n es el último elemento,

- La cantidad de elementos n es la longitud de la lista

Ejemplos

- [**1**, **2**, **3**] es la lista con los valores **1**, **2**, **3**
- [**1**, **-1**, **1**, **-1**, **1**, **-1**] es la lista que alterna 3 veces entre **1** y **-1**, empezando por **1**
- [↑, →, ↓, ←] es la lista de direcciones ordenada en el sentido de las agujas del reloj, empezando por ↑
- [Verdadero, Verdadero, Verdadero, Verdadero] es una lista con 4 Verdadero

- [] es la lista vacía

Ejemplos inválidos

¿Por qué los siguientes ejemplos son inválidos?

- [**1**, ↑, **2**, →]
- [→, Verdadero, Verdadero]
- La lista de todos los números pares.

Valores, expresiones (recordatorio)

- Valor: igual a sí mismo y distinto del resto
- Expresión: forma de denotar un valor
  - Distintas expresiones pueden denotar el mismo valor
  - La misma expresión puede denotar distintos valores
  - Se usan como argumentos y SOLO como argumentos

- Hay que definirlo antes de poder usar listas

Valores de Lista

- Valor: $[v_1, \ldots, v_n]$, donde $v_1, \ldots, v_n$ son valores del mismo tipo.
  - SI: [**1**, **2**, **3**], [↑], [], etc
  - NO: [**1**, ↑] por distintos tipos de elementos
  - NO: [`minColor()`] ya que `minColor()` no es un valor
- Notar: cada lista es igual a sí misma y distinta del resto de las listas (y valores)

- Como siempre, los valores no se pueden escribir en XGobstones
- Para denotar listas usamos expresiones.

Listas por extension

- Si $e_1, \ldots, e_n$ son expresiones del mismo tipo...
- ...y $v_1, \ldots, v_n$ son los valores denotados por $e_1, \ldots, e_n$...
- ...$[e_1, \ldots, e_n]$ es una expresión que $[v_1, \ldots, v_n]$

- `[1, 2, 3* 1]` denota la lista $[\mathbf{1}, \mathbf{2}, \mathbf{3}]$
- `[Norte, Este, Sur, Oeste]` denota la lista $[\uparrow, \rightarrow, \downarrow, \leftarrow]$
- `[siguiente(Azul), previo(Verde)]` denota la lista $[,]$

- ¿`[hayBolitas(Azul), hayBolitas(Rojo)]`?

Ejercicio I: hola mundo de las listas

- Escribir una función singleton que, dado un valor `x` retorne la lista [x].

```
1 function singleton(x) {
2     return([x])
3 }
```

- [x] es una expresión...
- ...ergo, como cualquier expresión, puede usarse como argumento y solo como argumento

Rangos

- En teoría, un rango `[<primero>..<ultimo>]` ES una lista:
  - es una colección de valores,
  - finita,
  - cuyos elementos son del mismo tipo.

- En la práctica también
- La expresión `[<primero>..<ultimo>]` denota la lista que:
  - en la primer posición tiene `<primero>`
  - en la segunda posicion tiene `siguiente(<primero>)`
  - ...
  - en la ultima posicion tiene `<ultimo>`

- `[1..3]` denota la lista [**1**, **2**, **3**]
- `[minDir()..maxDir()]` denota la lista [↑, →, ↓, ←]
- `[4..0]` denota la lista []

Ejercicio I: hola mundo de los rangos

- Escribir una función rangoAA que, dados dos valores `x` e `y`, retorne el rango de valores entre `x` e `y`, excluyendo `x` e `y`

```
1  function rangoAA(x,y) {
2    rango := []
3    if(x < y) {
4      rango := [siguiente(x)..previo(y)]
5    }
6    return (rango)
7  }
```

- Puedo tener variables que recuerden listas,
- y los rangos denotan listas que se pueden asignar a estas variables.

Concatenación de listas: motivación

- ¿Qué expresión denota una lista con 3 elementos ?
- ¿Qué expresión denota una lista con 10 elementos ?
- ¿Qué expresión denota una lista con 10000 elementos ?
- Necesitamos una forma de generar listas programáticamente, acumulando los valores en la lista

```
1  //Proposito: denota una lista con n repeticiones del valor v
2  function repeticion(n,v) {
3    lista := []
4    repeat(n) {
5      lista := ``Agregar v al final de lista''
6    }
7    return(lista)
8  }
```

Concatenación de listas: definición

función de concatenación (++)

- Si `ls` denota $[v_1, \ldots, v_n]$,
- `ms` denota $[w_1, \ldots, w_m]$,
- y v_1 y w_1 tienen el mismo tipo (o las listas son vacías), entonces

`ls ++ ms` denota la lista $[v_1, \ldots, v_n, w_1, \ldots, w_m]$.

- En otras palabras, `ls ++ ms` denota la lista que se obtiene de poner `ms` atrás de `ls`

- `[1,2,3] ++ [4..6]` denota $[\mathbf{1}, \mathbf{2}, \mathbf{3}, \mathbf{4}, \mathbf{5}, \mathbf{6}]$
- `[Este..maxDir()]++ [Norte]` denota la lista $[\rightarrow, \downarrow, \leftarrow, \uparrow]$
- `[] ++ [1]` denota $[\mathbf{1}]$

Concatenación: agregar adelante y agregar atrás

- Si `ls` es una lista y `e` denota un valor, entonces
  - `ls ++ [e]` denota la lista de agregar `e` atrás de `ls`
  - `[e] ++ ls` denota la lista de agregar `e` adelante de `ls`

- Notar que `ls ++ e` tiene un error de tipos
  - `e` no denota una lista

Concatenación de listas: uso

- Escribir una función `repeticion(n, v)` que retorne una lista con `n` repeticiones de `v`

```
1  //Proposito: denota una lista con n repeticiones del valor v
2  //Tipos: n n mero
3  function repeticion(n,v) {
4    acumulador := []
5    repeat(n) {
6      acumulador := acumulador ++ [v]
7      //acumulador := acumulador ++ v es un error de tipos
8    }
9    return(acumulador)
10 }
```

Ejercicio II

- Escribir una función `agregarSiHayColor` que, dada una lista de colores `ls` y un color `c`, devuelva la lista que se obtiene de agregar `c` al final de `ls` si en la celda actual hay bolitas de color `c`.

```
1  function agregarSiHayColor(ls, c) {
2    res := ls
3    if(hayBolitas(c)) {
4      res := ls ++ [c]
5    }
6    return(res)
7    //O: return(ls ++ singularSi(hayBolitas(c), c))
8  }
```

- ¿Por qué es necesaria la variable res?
- ¿Qué tiene de malo el nombre de la función?

Ejercicio III

- Escribir una función `colores` que denote la lista de colores que "aparecen" en las celda actual.

```
1  function colores() {
2    res := []
3    foreach c in [minColor()..maxColor()] {
4      res := agregarSiHayColor(res, c)
5      //O res := res ++ singularSi(hayBolitas(c), c)
6    }
7    return(res)
8  }
```

Ejercicio IV

- Escribir una función listaBolitas que, dado un color c, denote la lista que en la *n*-ésima posición tenga la cantidad de bolitas de color c que aparece en la n-ésima celda de un recorrido del tablero (elegir cualquier forma de recorrer el tablero).

```
1  function listaBolitas(c) {
2    res := []
3    IrAlInicioT(Norte, Este)
4    while(puedeMoverT(Norte, Este)) {
5      res := res ++ [nroBolitas(c)]
6      MoverT(Norte, Este)
7    }
8    return(res ++ nroBolitas(c))
9  }
```

Recorrido de listas con `foreach`

Repetición indexada (`foreach`)

```
foreach <indice> in <lista>
<bloque a repetir>
```

donde,

- *<indice>* es un identificador
- *<lista>* denota una lista

- Funciona con rangos porque, en realidad, los rangos son listas!

Recorrido de listas con `foreach`

- `foreach` ejecuta *<bloque a repetir>* una vez por cada elemento de la lista
- Esta ejecución es en orden
  - Primer repetición: corresponde al primer elemento de la lista
  - Segunda repetición: corresponde al segundo elemento de la lista
  - *n*-ésima repetición: corresponde al *n*-ésimo elemento de la lista
  - Última repetición: corresponde al último elemento de la lista
- *<indice>*: denota el valor correspondiente de la lista
  - Escribiendo su identificador se accede al valor denotado

Ejercicio I

- Escribir un procedimiento `RenderCamino` que, dada una lista de direcciones y un color, recorra cada una de las direcciones desde la celda actual, poniendo una bolita del color en cada celda. ¿Cuál es la precondición del procedimiento?

```
1 procedure RenderCamino(camino, color) {
2   foreach dir in camino {
3     Poner(color)
4     Mover(dir)
5   }
6 }
```

Ejercicio II

- Escribir un procedimiento `RenderColores` que, dada una lista de colores, recorra cada las celdas del tablero poniendo en la posición `n` una bolita del color que indique la posición `n` de la lista
  Suponer: la longitud de la lista es menor al tamaño del tablero

```
1  procedure RenderColores(colores) {
2    //Recorremos el tablero y la lista en paralelo
3    IrAlInicioT(Norte, Este)
4    foreach color in colores {
5      Poner(color)
6      MoverT(Norte, Este)
7    }
8  }
```

- ¿Cómo hacemos si la longitud de la lista es mayor o igual al tamaño del tablero?

Ejercicio III

- Escribir una función `incremento(ns, inc)` que, dada una lista de números `ns` y un número `n`, denote la lista que se obtiene de incrementar en `inc` cada elemento de la lista
  Ejemplo: `incremento([1,0,2,1], 3)` $\longrightarrow$ **[4, 3, 5, 4]**

```
1  function incremento(ns, inc) {
2    res := []
3    foreach n in ns {
4      res := res ++ [n + inc]
5    }
6    return(res)
7  }
```

Ejercicio IV

- Escribir una función `sinElemento(ls, elem)` que, dada una lista `ls` y un elemento `elem`, denote la lista que se obtiene de sacar todas las apariciones de `elem` en `ls`
  Ejemplo: `sinElemento([1,0,2,1], 1)` $\longrightarrow$ **[0, 2]**

```
1  function sinElemento(ls, elem)
2    res := []
3    foreach x in ls {
4      res := res ++ singularSi(x /= elem, x)
5    }
6    return(res)
7  }
```

Ejercicio V

- Escribir un procedimiento `PonerEscaleraDescenciente(n, c)` que, dado un número `n` y un color `c`, recorra el tablero y ponga `n` bolitas de color `c` en la primer celda, `n - 1` en la segunda, y asi siguiendo hasta que pone `1` bolita de color `c` en la celda `n`.

```
1  procedure PonerEscaleraDescenciente(n, c) {
2    IrAlInicioT(Norte, Este)}
3    PonerN(n, c)}
4    foreach i in reverso([1..n]) {
5      MoverT(Norte, Este)
6      PonerN(i, c)
7    }
8  }
```

- `foreach` es poderoso, porque la lista a recorrer puede ser resultado de una función.

Acceso a los elementos de una lista

- Funciones para denotar al primer y último elemento.
- Funciones para saber si una lista está vacía.
- Funciones para denotar la lista sin el primer o último elemento.

- Las funciones de lista sólo operan con el primer o último elemento.
- Para procesar elementos intermedios, deshacernos de los primeros/últimos

Funciones de lista

- Funciones de lista.
  - `ls` es una lista que denota $[v_1, \ldots, v_n]$
  - `tail(ls)` e `init(ls)` denotan listas nuevas.

`isEmpty(ls)`:	Denota VERDADERO cuando `ls` es `[]`
`head(ls)`:	Denota el primer elemento de `ls`, i.e., v_1 Precondición: `not isEmpty(ls)`
`last(ls)`:	Denota el último elemento de `ls`, i.e, v_n Precondición: `not isEmpty(ls)`
`tail(ls)`:	Denota la lista que se obtiene de sacar el primer elemento de `ls`, i.e., $[v_2, \ldots, v_n]$. Precondición: `not isEmpty(ls)`
`init(ls)`:	Denota la lista que se obtiene de sacar el último elemento de `ls`, i.e., $[v_1, \ldots, v_{n-1}]$. Precondición: `not isEmpty(ls)`

Ejercicio

¿Qué denota cada una de las siguientes expresiones?

- `isEmpty([])` $\longrightarrow$ Verdadero
- `isEmpty([2, 4])` $\longrightarrow$ Falso
- `head([2, 3])` $\longrightarrow$ **2**
- `head([])` $\longrightarrow$ BOOM
- `last([Norte, Este, Sur, Oeste])` $\longrightarrow \leftarrow$
- `last([])` $\longrightarrow$ BOOM
- `tail([2, 3, 4])` $\longrightarrow$ **[3, 4]**
- `init([2, 3, 4])` $\longrightarrow$ **[2, 3]**
- `head(tail(init([2, 3, 4])))` $\longrightarrow$ **3**

Funcionando ando I

- Escribir una función `rotacion` que, dada la lista que denota $[v_1, \ldots, v_n]$ denote la lista $[v_2, \ldots, v_n, v_1]$. En otras palabras, la lista que se obtiene de pasar el primer elemento hacia el final. ¿Cuál es la precondición de la función?

```
1  function rotacion(lista)
2    res := []
3    if(not isEmpty(lista))
4      res := tail(lista) ++ [head(lista)]
5    }
6    return(res)
7    //Lo siguiente no funciona:
8    //ifElse(isEmpty(res), [], tail(lista) ++ [head(lista)])
9    //Porque ifElse no tiene cortocircuito (es una funcion)
10 }
```

Funcionando ando II (en casa)

- Escribir una función `rotacionN` que, dada la lista y un numero `n`, rote `n` veces la lista como en la función anterior.

Funcionando ando III

- Escribir una función `sublistaHasta(ls, elem)` que, dada una lista `ls` y un valor `elem`, denote la sublista de `ls` desde el primer elemento hasta la primer aparición de `elem`
  Ejemplo: `sublistaHasta([1, 0, 2, 1], 2)` $\longrightarrow$ **[1, 0]**
  Ejemplo: `sublistaHasta([1, 0, 2, 1], 4)` $\longrightarrow$ BOOM

```
1  function sublistaHasta(ls, elem)
2    rec_ls := ls
3    sublista := []
4    while(head(rec_ls) /= elem) {
5      sublista := sublista ++ [head(rec_ls)]
6      rec_ls := tail(rec_ls)
7    }
8    return(sublista)
9  }
```

- Usar `foreach` no es conveniente, porque no queremos recorrer todos los elementos.

Recorrido de listas con `while`

- Esquema básico de recorrido para listas con `while`
- Procesa de primero a ultimo.

```
1  esquema Recorrido(lista, <params>) {
2    rec_lista := lista
3    while(not isEmpty(rec_lista) /* && it!<conds(args)>! */) {
4      <Procesar>(head(rec_lista), <args>)
5      rec_lista := tail(rec_lista)
6    }
7  }
```

Recorrido de listas con `while`

- Esquema básico de búsqueda en listas
- Busca de primero a ultimo

```
1  esquema Busqueda(lista, <params>)
2    rec := lista
3    while(/*not isEmpty(rec) && */
4          not <esElBuscado>(head(rec), <args>)) {
5      rec := tail(rec)
6    }
7    return(/*not isEmpty(rec) && */ <esElBuscado>(head(rec), <args>))
8  }
```

Ejercicio

- Reescribir usando `while` la función `incremento(ns, inc)` que, dada una lista de números `ns` y un número `n`, denote la lista que se obtiene de incrementar en `inc` cada elemento de la lista

```
1  function incremento(ns, inc) {
2    res := []
3    rec_ns := ns
4    while(not isEmpty(rec_ns)) {
5      res := res ++ [head(rec_ns) + inc]
6      rec_ns := tail(rec_ns)
7    }
8    return(res)
9  }
```

`foreach` VS. `while`

- `foreach` es más simple porque se encarga del recorrido
- `while` es más flexible porque puedo controlar cómo se recorre

- Si es necesario recorrer todos los elementos exactamente una vez, conviene `foreach`
- Si se recorre sólo una parte (inicio/fin) de los elementos, `while` es más simple

- Se pueden combinar los dos esquemas cuando se recorre más de una lista
- La clase que viene se ven muchos ejemplos. . .

Tipos de listas

- En XGobstones (y Gobstones) todo valor tiene un tipo...
- ...que determina cuándo puede ser argumento de una funcion/procedimiento

- Para "concatenar" dos listas, sus elementos deben tener el mismo tipo
- Las listas se distinguen, pues, por el tipo de sus elementos
  - Lista de números: cuando sus elementos son números
  - Lista de booleanos: cuando sus elementos son booleanos
  - etc.

- Una lista puede a la vez contener listas
  - Las listas son valores, ergo, puedo meterlas en una lista.
  - Tipos: Lista de lista de números, Lista de lista de lista de números, etc
- XGobstones tiene infinitos tipos

Tipos de listas: lista vacía

- ¿Cuál es el tipo de una lista sin elementos?
- Tiene que poder "concatenarse" con listas de cualquier tipo
- Entonces, [] tiene infinitos tipos, ya que tiene tipo...
- ...lista de T para cualquier tipo T
  - [] tiene tipo lista de número,
  - [] tiene tipo lista de direcciones,
  - [] tiene tipo lista de lista de números,
  - [] etc.
- Cuando se usa [], el valor asume el tipo que le conviene

- Decimos que [] tiene tipo lista de $*$

Ejercicio de tipos

- ¿Qué tipo tienen las siguientes expresiones que denotan listas?

- `[1, 2]` $\longrightarrow$ lista de números
- `[hayBolitas(Rojo)]` $\longrightarrow$ lista de booleanos
- `[]` $\longrightarrow$ lista de $*$
- `[[1, 2], [0]]` $\longrightarrow$ lista de lista de números
- `[[1, 2], []]` $\longrightarrow$ lista de lista de números
- `[[], [], []]` $\longrightarrow$ lista de lista de $*$
- `[[[1]], [], []]` $\longrightarrow$ lista de lista de lista de número
- `[[[]], [], []]` $\longrightarrow$ lista de lista de lista de $*$

Clase que viene

Vamos a hacer ejercicios que incluyan:

- Funciones de listas sin utilizar tableros.
- Procesamientos usuales sobre listas:
  - búsqueda, filtro, transformación, mezcla.
- Algún procesamiento inusual sobre listas.
- Pizca de Listas de Listas, quizá

Hoy va a estar la nueva práctica