Algoritmos y Estructuras de Datos II

Tipos concretos

3 de abril de 2017

Tipos de datos

- Arreglos
- Listas
- Tuplas
- Punteros

Ejemplos

Tipos de datos

Conceptualmente distinguimos dos clases de tipos de datos:

- Tipos de datos **concretos**:
  - son provistos por el lenguaje de programación,
  - es un concepto **dependiente** del lenguaje de programación,
  - comúnmente: enteros, char, string, booleanos, arreglos, reales,
- Tipos de datos **abstractos**:
  - **surgen de** analizar **el problema** a resolver,
  - es un concepto **independiente** del lenguaje de programación,
  - eventualmente se implementará utilizando tipos concretos,
  - eso da lugar a una **implementación** o **representación** del tipo abstracto
  - ejemplo: si se quiere desarrollar una aplicación para un gps que calcule ciertos caminos óptimos, seguramente surgirá considerar un grafo donde las aristas son segmentos de rutas.

Tipos abstractos de datos (TADs)

- Identificar los tipos abstractos de datos **surge de analizar detenidamente el problema a resolver**.
- Identificarlos y especificarlos es una tarea que **siempre es recompensada**.
- Ejemplificaremos a través de una serie de problemas, cada uno de ellos dará lugar a un tipo abstracto.
- Pero antes (clase de hoy) hablaremos de tipos concretos habituales.
- Saltearemos tipos sencillos conocidos: booleanos, enteros, char, reales.
- Abordaremos tipos más complejos, como tuplas, listas, arreglos.

Arreglos

Declaración

- La mayoría de los lenguajes de programación proporcionan arreglos como tipo concreto.
- Dado un tipo **T**, se declaran, por ejemplo, de la forma:

  type tarray = **array**[M..N] **of T**

  var a: tarray
- El tipo tarray así definido corresponde al producto Cartesiano $\mathbf{T}^{\text{N-M+1}}$.
- Las celdas de los arreglos se alojan normalmente en **espacios contiguos** de memoria.
- Algunos lenguajes, como C, imponen que M debe ser 0 (y por lo tanto no hace falta escribirlo).
- En el teórico-práctico no adoptamos esa imposición.

Arreglos
Índices

- Por el contrario, podemos permitirnos más libertad.
- Por ejemplo, otra posibilidad es:

  type tindex = **array**['a'..'z'] **of nat**
  var page: tindex
- El arreglo page podría servir de índice en un listado de un padrón electoral, por ejemplo, informando en qué página del padrón aparecen listadas las personas cuyos nombres comienzan con cada letra. Por ejemplo, page['g'] = 271 significaría que en la página 271 del padrón comienzan los nombres de personas cuya primera letra es la letra g.

Arreglos
Índices

- Por ejemplo, otra posibilidad es:

  type tweek = (sun, mon, tue, wed, thu, fri, sat)
  type tcalendar = **array** [mon..fri] **of** **T**
  var cal: tcalendar

- El arreglo cal posee 5 celdas, una para cada día hábil de la semana. Por ejemplo, con un **T** adecuado, cal podría almacenar las tareas a desarrollar cada uno de esos días.

Arreglos

Índices

- Esto muestra que se puede utilizar una variedad de conjuntos como índices de arreglos.
- Debe haber una clara noción de **el siguiente de**,
  - cosa que ocurre con enteros (el siguiente de 4 es 5),
  - caracteres (el siguiente de 'h' es 'i')
  - tipos enumerados como tweek (el siguiente de wed es thu),
  - entre otros.

Arreglos

Dimensiones

- También es frecuente la utilización de arreglos multidimensionales,
- ejemplos:

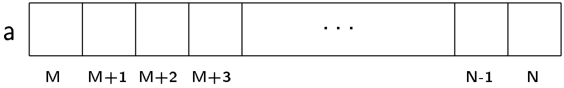
  type tarray1 = **array**[1..N,1..M] **of T**
  type tarray2 = **array**[1..N,'a'..'z',sunday..saturday] **of T**
  var b: tarray1
  var c: tarray2

- Los arreglos bidimensionales se suelen denominar matrices y se grafican como tales.
- Para enfatizar el hecho de que un arreglo es unidimensional a veces se lo llama vector.

Arreglos

Representación gráfica

- Las celdas de un arreglo suelen alojarse en espacios contiguos de memoria.
- Por ello, el arreglo a declarado recientemente suele representarse gráficamente de la siguiente manera:

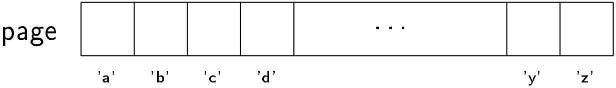


- Se observa una celda para cada índice entre M y N.
- Al desplegarse en forma adyacente sugiere efectivamente que se alojan en espacios contiguos de memoria.

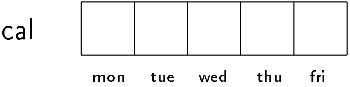
Arreglos

Representación gráfica

- El arreglo page, declarado anteriormente puede representarse gráficamente de la siguiente manera:



- mientras que el arreglo cal, puede representarse así

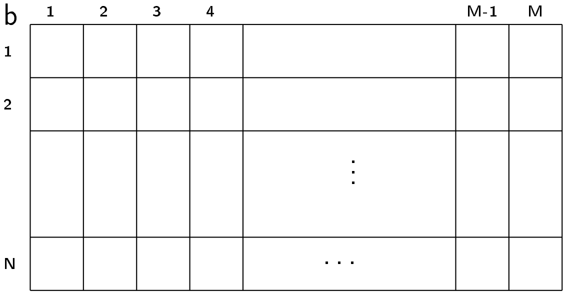


- Cualquiera de los arreglos unidimensionales (a, page, cal, etc.) puede representarse también verticalmente.

Arreglos

Representación gráfica de arreglos bidimensionales

- Los arreglos bidimensionales se denominan también matrices, y se representan gráficamente como tales.
- Por ejemplo, el arreglo b declarado anteriormente puede representarse gráficamente de la siguiente manera:



Arreglos

Operaciones

- El valor alojado en la celda i de a se obtiene evaluando la expresión a[i] y se modifica asignando a a[i].
- Ejemplo de acceso al valor alojado en dicha celda: x:= a[i] + 3 (si a es, por ejemplo, un arreglo de naturales)
- Ejemplo de modificación de dicha celda a[i]:= 7.
- Otros ejemplos:
  - a[i]:= i
  - a[i]:= a[j]
  - a[a[i]]:= a[i]
- Similarmente para los otros arreglos (asumiendo que T es **nat**)
  - k:= page['a']+1
  - t:= cal[fri]
  - b[i,k] := b[i,j] + b[j,k]
  - En b[i,k], intuitivamente i indica la fila y k la columna.

Arreglos

Orden

- Al estar alojado en espacios contiguos, con una cuenta muy sencilla el programa puede calcular donde se encuentra cada celda.
- Por eso, acceder o modificar cualquier celda lleva tiempo *constante*.
- Es decir, el tiempo de acceso al valor de una celda, o el tiempo de modificación de una celda no depende del número de celdas
- (pero sí puede depender del tipo **T** ya que lo que se está accediendo o modificando es un elemento de ese tipo).
- O sea que hicimos bien en elegir comparaciones como operación elemental al analizar algoritmos de ordenación.

Arreglos

Tamaño

- Los arreglos tienen longitud prefijada: en el caso del arreglo a, N-M+1.
- Normalmente N $>$ M, pero se puede admitir también N$=$M (longitud 1) e incluso N $<$ M (longitud 0).
- El tamaño total del arreglo (espacio ocupado en memoria) es la longitud del mismo multiplicada por el tamaño de cada celda, que depende del tipo **T**.
- Es decir, el espacio que ocupa es *del orden de n* donde *n* es la longitud del arreglo (diremos que el espacio que ocupa es *lineal* en el número de celdas).

Arreglos

Comando **for**

- Para inicializar y modificar arreglos es muy común utilizar el comando **for**.
- Por ejemplo, el siguiente comando inicializa todas las celdas de a con el valor 0.

  for i:= M **to** N **do** a[i]:= 0 **od**
- Intuitivamente, el comando dice que para todo valor de i entre M y N, ambos inclusive, se asigna 0 a la celda a[i].

Arreglos
Comando **for**

- El comando **for** adquiere en general la forma

  for i:= M **to** N **do** c **od**
  for k:= 'a' **to** 'z' **do** c **od**
  for d:= tue **to** fri **do** c **od**

- donde en los tres ejemplos, c, llamado **el cuerpo** del **for**, es cualquier comando que **no modifica** el valor de la variable que se usa como índice[1] (i, l y d respectivamente en estos ejemplos).

[1] Lamentablemente, la mayoría de los lenguajes de programación permiten que dicha variable sea modificada en el cuerpo del **for**. Se considera una **muy mala práctica** de programación escribir un **for** en el que eso ocurre.

Arreglos

Comando **for**

En el ejemplo del arreglo tridimensional c declarado más arriba, si se quiere inicializar todo el arreglo (asumiendo que **T** es, por ejemplo, **nat**), se lo puede hacer a través de 3 ciclos **for** anidados:

```
for i:= M to N do
    for k:= 'a' to 'z' do
        for d:= tue to fri do
            c[i,k,d]:= 0
        od
    od
od
```

Arreglos

Comando **for**

- Por último, decimos que un invariante de

  for i:= M **to** N **do** c **od**

  es un predicado $\mathcal{I}(i)$, tal que
  - $\mathcal{I}$(M) vale antes de la ejecución del **for**,
  - y la validez de $\mathcal{I}(i) \wedge M \leq i \leq N$ antes de cada ejecución de c garantiza la validez de $\mathcal{I}(i+1)$ después de dicha ejecución de c.
- Entonces, si N $\geq$ M-1, al finalizar el **for** se cumple $\mathcal{I}$(N+1).

Arreglos

Comando **for**

- A veces alcanza con un **for** más abstracto: cuando no resulte necesario mencionar las posiciones y sólo interesen los valores alojados en las celdas. Por ejemplo,

  s := 0
  for e $\in$ a **do** s:= s + e **od**

  suma todos los elementos del arreglo a, sin importar cuántas dimensiones tiene. Esta notación tampoco revela claramente el orden en que se procesan los elementos del arreglo.

Listas

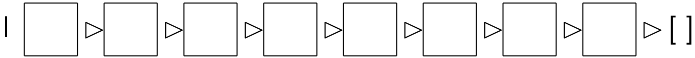
Declaración

- Algunos lenguajes de programación (por ejemplo Haskell) permiten declarar listas:
- **type** tlist = **list of T**
  var l: tlist
- El tipo tlist así definido corresponde a la unión de los productos Cartesianos $\mathbf{T}^i$, es decir, corresponde a $\bigcup_{i=0}^{\infty} \mathbf{T}^i$.
- Así, a diferencia del arreglo cuya longitud está predeterminada, el número de elementos de una lista no lo está.
- A priori, puede contener una cantidad arbitraria de elementos de **T**.
- Las celdas de la lista **no** necesariamente se alojan en **espacios contiguos** de memoria.

Listas

Representación gráfica

- La lista l declarada recientemente puede representarse gráficamente de la siguiente manera:



- En la representación gráfica se ve una celda para cada elemento de la lista, que al desplegarse con el símbolo ▷ sugiere la existencia de una secuencia de celdas: cada celda señala la siguiente.

Listas

Operaciones

- Se puede acceder a un elemento de la lista o modificarlo a través de la operación "."
- Por ejemplo, k:= l.i ó l.i:= 5. Esto es muy parecido a lo que ocurre con los arreglos.
- Además, puede modificarse la propia lista, por ejemplo
  - l:= e $\triangleright$ l agrega un elemento al comienzo y
  - l:= tail(l) lo quita.
- Son justamente estas operaciones especiales las que dificultan alojar una lista en espacios contiguos de memoria.
- Dada una lista l es posible calcular la longitud #(l) de la misma.

Listas

Operaciones

- Al agregarse un elemento a una lista, no hay ninguna garantía de que haya espacio libre justo en la posición de memoria adyacente a donde se encuentra el primer elemento de la lista.
- Si se quisiera alojar la nueva lista en espacios contiguos habría que copiar la lista entera en una parte de la memoria donde haya suficiente espacio libre para toda la lista.
- Esto no es lo que habitualmente se hace (salvo destacables excepciones) ya que las modificaciones requerirían copiar toda la lista y por lo tanto serían *lineales* en el número de celdas.
- En lugar de esto, se aloja el elemento que se quiere agregar en una nueva posición de memoria (cualquiera que esté libre) y se deja un registro que indica en qué posición de la memoria se encuentran los siguientes elementos de la lista.

Listas

Orden de acceso

- Así, estas modificaciones resultan *constantes* en lugar de *lineales*, o sea, no dependen del número de celdas de la lista.
- Calcular la longitud puede ser constante o lineal según la implementación.
- Luego de una secuencia de modificaciones, los elementos de una lista pueden quedar desperdigados en la memoria.
- Siempre se puede recorrer la lista ya que se cuenta con la información necesaria, como sugiere la representación gráfica, para ir de cada elemento de la lista al siguiente.
- Esto significa que para acceder al i-ésimo elemento de una lista es necesario recorrerla secuencialmente a partir del primero hasta encontrarlo, operación que resulta *del orden de i*.

Listas

Comando **for**

- No siempre los lenguajes de programación tienen el tipo concreto lista.
- Los más importantes ofrecen, sin embargo, alguna forma de implementarlo. Veremos luego que se pueden implementar listas usando los tipos concretos tupla y puntero.
- Puede convenir a veces extender la notación del **for** para recorrer listas.
- Por ejemplo, si se quiere ejecutar c una vez para cada elemento e de la lista l (del primero al último) se escribe:

  s:= 0
  for i:= 0 **to** #(l)-1 **do** s:= s + l.i **od**

Listas

Comando **for**

- La misma versión abstracta del **for** que utilizamos para arreglos puede utilizarse para listas cuando el cuerpo del **for** no necesita referirse a las posiciones. Por ejemplo,

  s := 0
  for e $\in$ l **do** s:= s + e **od**

  suma todos los elementos de la lista l, igual que el ejemplo de la filmina anterior.

Tuplas

Declaración

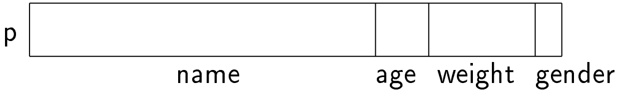
También llamados registros o estructuras, se utilizan para representar productos Cartesianos, pero ahora cuando los conjuntos entre los que se hace el producto son distintos, es decir, de la forma $\mathbf{T}_1 \times \mathbf{T}_2 \times \mathbf{T}_3$ donde los $\mathbf{T}_i$ pueden ser tipos distintos. Se declaran de la siguiente forma

```
type tperson = tuple
                  name: string
                  age: nat
                  weight: real
                  gender: (male, female)
               end tuple
var p: tperson
```

Tuplas

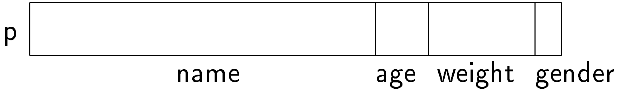
Representación gráfica

El tipo tperson así definido corresponde a **string** $\times$ **nat** $\times$ **real** $\times$ $\{$male, female$\}$, y name, age, weight y gender se llaman **campos**. Las tuplas se alojan normalmente en **espacios contiguos** de memoria. Se puede representar gráficamente de la siguiente manera:



Tuplas

Representación gráfica



- En la misma se ven los campos de distinto tamaño porque cada uno de ellos puede ocupar un espacio diferente.
- Lo alojado en cada campo se obtiene evaluando las expresiones p.name, p.age, p.weight y p.gender y se modifica de manera similar (por ejemplo, p.name:= "Juan').
- Al estar alojado en espacios contiguos de memoria acceder o modificar cualquier campo lleva tiempo **constante**, aunque depende del tipo $\mathbf{T}_i$ del campo en cuestión.
- El espacio de memoria que ocupa una tupla es la suma de los espacios que ocupan sus campos.

Punteros

Declaración

Dado un tipo T, se puede declarar el tipo "puntero a T". Por ejemplo, si tperson es el tipo definido más arriba

type tp_person = **pointer to** tperson
var p: tp_person

La variable p así declarada es un puntero a tperson. Esto significa que p puede almacenar una dirección de memoria donde se aloja una tperson.

Punteros

Operaciones

Las operaciones con punteros son las siguientes:

p:= e
alloc(p)
free(p)

- El primer comando es una asignación: e es una expresión cuyo valor es la dirección de memoria de una tperson, la asignación tiene por efecto que dicha dirección sea alojada ahora en p.
- El segundo comando reserva un nuevo espacio de memoria capaz de almacenar una tperson, y la dirección de ese nuevo espacio de memoria se aloja en p.

Punteros

Operaciones

Las operaciones con punteros son las siguientes:

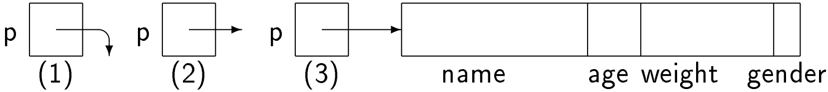
p:= e
alloc(p)
free(p)

- El tercer comando libera el espacio de memoria señalado por p, es decir, cuya dirección se encuentra alojada en p.
  - Puede darse que p tenga como valor una dirección de memoria que no está actualmente reservada (por ejemplo, inmediatamente después de haber ejecutado free(p)).
  - Para evitar permanecer en este estado, existe un valor especial que puede adoptar un puntero, llamado **null**.
  - Cuando el valor de p es **null**, p no señala ninguna posición de memoria.

Punteros

Representación gráfica

Hay distintas representaciones gráficas, una para cada una de las posibles situaciones:



- En la situación (1), el valor de p es **null**, p no señala ninguna posición de memoria.
- En la situación (2) la posición de memoria señalada por p no está reservada, por ejem. inmediatamente después de free(p).
- En la situación (3) el valor de p es la dirección de memoria donde se aloja la tperson representada gráficamente al final de la flecha, por ejemplo, inmediatamente después de alloc(p).

Punteros

Representación gráfica

- En la situación (3), $\star$p denota la tperson que se encuentra señalada por p, y por lo tanto, $\star$p.name, $\star$p.age, $\star$p.weight y $\star$p.gender, sus campos.
- Esta notación permite acceder a la información alojada en la tperson y modificarla mediante asignaciones a sus campos (por ejemplo, $\star$p.name:= "Juan").

Punteros

Notación

- Una notación conveniente para acceder a los campos de una tupla señalada por un puntero es la flecha "$\rightarrow$".
- Así, en vez de escribir $\star$p.name, podemos escribir p$\rightarrow$name tanto para leer ese campo como para modificarlo.
- Esta notación reemplaza el uso de dos operadores ("$\star$" y ".") por uno visualmente más apropiado (por ejemplo, p$\rightarrow$name:= "Juan").
- La notación $\star$p y sus derivadas $\star$p.name, p$\rightarrow$name, etc. sólo pueden utilizarse en la situación (3).

Punteros

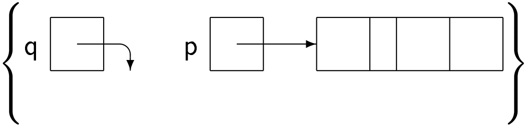
Punteros colgantes y **null**

En la situación (2) el valor de p es inconsistente, no debe utilizarse ni accederse una dirección de memoria no reservada ya que no se sabe, a priori, qué hay en ella (en particular puede haber sido reservada para otro uso y al modificarlo se estaría corrompiendo información importante para tal uso). Los punteros que se encuentran en la situación (2) se llaman comúnmente referencias o punteros colgantes (dangling pointers).
En la situación (1) el valor de p es **null**, es decir que p no señala ninguna posición de memoria. Por ello, no tiene sentido intentar acceder a ella.

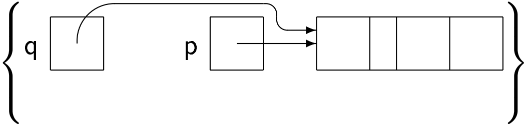
Punteros

Aliasing

Como vemos, los punteros permiten manejar explícitamente direcciones de memoria. Esto no es sencillo, aparecen situaciones que con los tipos de datos usuales no se daban. Por ejemplo:



q:=p



Punteros

Aliasing

Como se ve, después de la asignación, q y p señalan a la misma tupla, por lo que cualquier modificación en campos de $\star$q también modifican los de $\star$p (claro, ya que son los mismos) y viceversa. Estamos en presencia de lo que se llama **aliasing**, es decir, hay 2 nombres distintos ($\star$p y $\star$q) para el mismo objeto y al modificar uno se modifica el otro. Programar correctamente en presencia de aliasing es muy delicado y requiere gran atención.

Punteros

Orden

Acceder o modificar lo señalado por un puntero es claramente *constante*, ya que el puntero contiene la dirección exacta donde se encuentra en la memoria. El *orden* de las operaciones alloc y free, en cambio, depende del compilador del lenguaje. Existen diferentes maneras -no triviales- de implementarlas.

Punteros

Administración de la memoria

Siempre hemos asumido que no es necesario ocuparse de reservar y liberar espacios de memoria para las variables. Los punteros como p y q son variables, así que **tampoco es necesario reservar y liberar espacio para ellos**. Pero las operaciones alloc y free son las responsables de reservar y liberar explícitamente espacio **para los objetos que p y q señalan**.

Esta posibilidad significa ciertas libertades: el programador puede decidir exactamente cuándo reservar espacio para una tupla. Por otro lado, significa también más responsabilidad: el programador es el que debe encargarse de liberar el espacio cuando deje de ser necesario.

Punteros

Administración de la memoria

Pero el verdadero beneficio de los punteros radica en que permiten una gran flexibilidad para representar estructuras complejas, y por lo tanto, para implementar diferentes tipos abstractos de datos.

Creación de un arreglo

fun mk_array (n : **nat**) **ret** a: **array**[1..n] **of nat**
    for i:= 1 **to** n **do** a[i]:= i **od**
end fun

Creación de una lista

fun mk_list (n : **nat**) **ret** b: **list of nat**
    b:= [ ]
    for i:= n **downto** 1 **do** b:= i $\triangleright$ b **od**
end fun

Creación de una lista enlazada

```
type node = tuple
              value: nat
              next: pointer to node
            end tuple
type list = pointer to node
fun mk_linked_list (n : nat) ret c: list
    var aux: pointer to node
    c:= null
    for i:= n  downto  1  do
            alloc(aux)
            aux→value:= i
            aux→next:= c
            c:= aux
    od
end fun
```

Algoritmos y Estructuras de Datos II

Tipos Abstractos de Datos (TADs o ADTs en inglés)

5 de abril de 2017

Clase de hoy

Tipos abstractos de datos (TADs)

- Surgen de analizar el problema a resolver.
- Plantearemos un problema.
- Lo analizaremos.
- Obtendremos un TAD.

Paréntesis balanceados

- Problema:
  - Dar un algoritmo que tome una expresión,
  - dada, por ejemplo, por un arreglo de caracteres,
  - y devuelva verdadero si la expresión tiene sus paréntesis correctamente balanceados,
  - y falso en caso contrario.

Solución conocida

- Recorrer el arreglo de izquierda a derecha,
- utilizando un entero **inicializado en 0**,
- **incrementarlo** cada vez que se encuentra un paréntesis que abre,
- **decrementarlo** (comprobando previamente que no sea nulo en cuyo caso **no están balanceados**) cada vez que se encuentra un paréntesis que cierra,
- Al finalizar, **comprobar** que dicho entero sea cero.
- ¿Es necesario que sea un entero?

Contador

- No hace falta un entero (susceptible de numerosas operaciones aritméticas),
- sólo se necesita **algo** con lo que se pueda
  - inicializar
  - incrementar
  - comprobar si su valor es el inicial
  - decrementar si no lo es
- Llamaremos a ese **algo**, **contador**
- Necesitamos un contador.

TAD Contador

- El contador se define por lo que sabemos de él: sus cuatro operaciones
  - inicializar
  - incrementar
  - comprobar si su valor es el inicial
  - decrementar si no lo es
- Notamos que las operaciones **inicializar** e **incrementar** son capaces de generar todos los valores posibles del contador,
- **comprobar** en cambio solamente examina el contador,
- **decrementar** no genera más valores que los obtenibles por **inicializar** e **incrementar**
- A las operaciones **inicializar** e **incrementar** se las llama **constructores**

Especificación del TAD Contador

```
module TADContador where

data Contador = Inicial
              | Incrementar Contador

es_inicial :: Contador → Bool
decrementar :: Contador → Contador
- - se aplica solo a un Contador que no sea Inicial

es_inicial Inicial = True
es_inicial (Incrementar c) = False

decrementar (Incrementar c) = c
```

Especificación del TAD Contador (con colores)

```
module TADContador where

data Contador = Inicial
              | Incrementar Contador

es_inicial :: Contador → Bool
decrementar :: Contador → Contador
- - se aplica solo a un Contador que no sea Inicial

es_inicial Inicial = True
es_inicial (Incrementar c) = False

decrementar (Incrementar c) = c
```

Explicación

Los valores posibles del contador están expresados por

- Inicial
- Incrementar Inicial
- Incrementar (Incrementar Inicial)
- Incrementar (Incrementar (Incrementar Inicial))
- etcétera, es una lista infinita, pero cada uno tiene una cantidad finita de veces el constructor **Incrementar** aplicado al constructor **Inicial**

Intuitivamente

Intuitivamente estos valores se corresponden con números naturales:

- Inicial $\longrightarrow$ 0
- Incrementar Inicial $\longrightarrow$ 1
- Incrementar (Incrementar Inicial) $\longrightarrow$ 2
- Incrementar (Incrementar (Incrementar Inicial)) $\longrightarrow$ 3
- etcétera.

Intuitivamente

Una intuición más interesante es que cada **Incrementar** corresponde a agregar “una marquita” y cada **decrementar**, a borrarla:

- Inicial $\longrightarrow$
- Incrementar Inicial $\longrightarrow |$
- Incrementar (Incrementar Inicial) $\longrightarrow ||$
- Incrementar (Incrementar (Incrementar Inicial)) $\longrightarrow |||$
- etcétera.

Formalismo

Pero éstas son sólo intuiciones, formalmente los valores están expresados como dijimos antes, por

- Inicial
- Incrementar Inicial
- Incrementar (Incrementar Inicial)
- Incrementar (Incrementar (Incrementar Inicial))
- etcétera.

Podés verlo en Haskell en el archivo TADContador.hs.

Operaciones que no son constructores

- Observar que la operación **es_inicial** examina si su argumento es el primero de esta lista o no,
- y que la operación **decrementar** aplicado a cualquiera de esta lista (salvo el primero), devuelve el que se encuentra inmediatamente arriba
- no construyen valores nuevos,
- las operaciones **es_inicial** y **decrementar** no son constructores.

Operación **es_inicial**

Esta operación está definida por las ecuaciones

es_inicial : Contador $\rightarrow$ Bool
es_inicial (Inicial) = True
es_inicial (Incrementar c) = False

Ejemplos:

- es_inicial Inicial = True
- es_inicial (Incrementar Inicial) = False
- es_inicial (Incrementar (Incrementar Inicial)) = False
- etcétera.

Operación **decrementar**

Esta operación está definida por las ecuaciones

decrementar : Contador $\rightarrow$ Contador
{se aplica sólo a un Contador que no sea Inicial}
decrementar (Incrementar c) = c

Ejemplos:

- decrementar Inicial no satisface la pre-condición.
- decrementar (Incrementar Inicial) = Inicial
- decrementar (Incrementar (Incrementar Inicial) = Incrementar Inicial

Sobre la especificación

- Los **constructores** (en este caso **Inicial** e **Incrementar**) deben ser capaces de generar todos los valores posibles del TAD.
- En lo posible cada valor debe poder generarse de manera única.
- Esto se cumple para Inicial e Incrementar: partiendo de Inicial y tras sucesivos incrementos se puede alcanzar cualquier valor posible; y hay una única forma de alcanzar cada valor posible de esa manera.
- Las **demás operaciones** se listan más abajo.

Sobre las ecuaciones

- Las operaciones que no son constructores, deben definirse por ecuaciones
- Las ecuaciones deben considerar todos los casos posibles que satisfagan la precondición
- Ejemplo, las ecuaciones para la operación **es_inicial** considera los únicos dos casos posibles,
- Ejemplo, la ecuación para la operación **decrementar** considera el único caso posible.

Sobre las ecuaciones de **es_inicial**

- ¿Cómo nos convencemos de que las ecuaciones de **es_inicial** cubren todos los casos posibles?
- Comenzamos escribiendo

  es_inicial : Contador $\rightarrow$ Bool
  es_inicial c = ¿?

  donde c es una variable que representa un contador arbitrario.
- Pero no sabemos qué escribir en la parte derecha porque el resultado depende del valor de c.

Sobre las ecuaciones de **es_inicial**

- Como c representa un contador arbitrario, la reemplazamos por cada uno de los casos posibles de contadores: los construidos por Inicial y los construidos por Incrementar:

  es_inicial : Contador → Bool
  es_inicial Inicial = ¿?
  es_inicial (Incrementar c) = ¿?

- Ahora sí estamos en condiciones de saber cuál debe ser el resultado en cada caso:

  es_inicial : Contador → Bool
  es_inicial Inicial = True
  es_inicial (Incrementar c) = False

Sobre las ecuaciones de **decrementar**

- Lo mismo podemos hacer para **decrementar**:
- Comenzamos escribiendo

  decrementar : Contador $\rightarrow$ Contador
  decrementar c = ¿?

  donde c es una variable que representa un contador arbitrario **salvo Inicial**.
- A pesar de eso, no sabemos qué escribir en la parte derecha si no miramos quién es c.

Sobre las ecuaciones de **decrementar**

- Como c representa un contador arbitrario **salvo Inicial**, la reemplazamos por cada uno de los casos posibles de contadores: como el construido por Inicial no puede ser, quedan sólo los construidos por Incrementar:

  decrementar : Contador $\rightarrow$ Contador
  decrementar (Incrementar c) = ¿?

- Ahora sí estamos podemos completar el resultado:

  decrementar : Contador $\rightarrow$ Contador
  decrementar (Incrementar c) = c

Sobre las ecuaciones de **decrementar**

- Otra posibilidad sería generar los dos casos:

  decrementar : Contador $\rightarrow$ Contador
  decrementar Inicial = ¿?
  decrementar (Incrementar c) = ¿?

- y luego, o bien eliminamos el primero (y terminamos igual que en la filmina anterior),
- o bien, completamos informando que se trata de un error
- Ahora sí estamos podemos completar el resultado:

  decrementar : Contador $\rightarrow$ Contador
  decrementar Inicial = error "No se puede ..."
  decrementar (Incrementar c) = c

Prototipo

- Un prototipo es un primer ejemplo de solución, que permite comprobar el funcionamiento del producto futuro tempranamente (e introducir eventuales modificaciones antes de que sea tarde).
- Con la especificación, habitualmente se puede hacer rápidamente un prototipo.
- Podés verlo en Haskell en el archivo EjemplosContador.hs.

Resolviendo el problema

- Luego de obtenerse el prototipo, se quiere implementar un algoritmo que resuelve el problema utilizando una implementación del contador.
- Asumimos que el TAD Contador se implementará bajo el nombre **counter**,
- que habrá un procedimiento llamado **init** que implemente el constructor Inicial,
- uno llamado inc que implemente el constructor Incrementar,
- y uno llamado dec que implemente la operación decrementar.
- Habrá también una función **is_init** que implemente la operación es_inicial.

Especificación e implementación

- Utilizaremos nombres en castellano para constructores y operaciones especificadas,
- y nombres en inglés para sus implementaciones.
- Vamos a utilizar informalmente la notación $c \sim C$ para indicar que c implementa C.

Especificación e implementación

type counter = ... {- no sabemos aún cómo se implementará -}

proc init (**out** c: counter) {Post: c $\sim$ Inicial}

{Pre: c $\sim$ C} **proc** inc (**in**/**out** c: counter) {Post: c $\sim$ Incrementar C}

{Pre: c $\sim$ C $\wedge\neg$is_init(c)}
proc dec (**in**/**out** c: counter)
{Post: c $\sim$ decrementar C}

fun is_init (c: counter) **ret** b: **bool** {Post: b $=$ (c $\sim$ Inicial)}

Algoritmo de control de paréntesis balanceados

```
fun matching_parenthesis (a: array[1..n] of char) ret b: bool
    var i: nat
    var c: counter
    b:= true
    init(c)
    i:= 1
    do i ≤ n ∧ b → if a[i] = '(' → inc(c)
                        a[i] = ')' ∧ is_init(c) → b:= false
                        a[i] = ')' ∧ ¬is_init(c) → dec(c)
                        otherwise → skip
                     fi
                     i:= i+1
    od
    b:= b ∧ is_init(c)
end fun
```

Paréntesis balanceados: comentarios finales

- Luego veremos cómo implementar contadores.
- Condiciones e invariantes fueron omitidos por cuestiones de espacio,
- pero están en los apuntes.

Generalización de paréntesis balanceados

- Problema:
  - Dar un algoritmo que tome una expresión,
  - dada, por ejemplo, por un arreglo de caracteres,
  - y devuelva verdadero si la expresión tiene sus paréntesis, corchetes, llaves, etc. correctamente balanceados,
  - y falso en caso contrario.

Usando contadores

- ¿Alcanza con un contador?
  - "(1+2)"
  - "{1+(18-[4*2])}"
  - "(1+2}"
- ¿Alcanza con tres (o n) contadores?
  - "(1+2}´´
  - "(1+[3-1)+4]´´

Conclusión

- No alcanza con saber cuántos delimitadores restan cerrar,
- también hay que saber en qué orden deben cerrarse,
- o lo que es igual
- en qué orden se han abierto,
- mejor dicho,
- ¿cuál fue el último que se abrió? (de los que aún no se han cerrado)
- ¿y antes de ése?
- etc.
- Hace falta una “constancia” de cuáles son los delimitadores que quedan abiertos, y en qué orden deben cerrarse.

Solución posible

- Recorrer el arreglo de izquierda a derecha,
- utilizando dicha “constancia” de delimitadores aún abiertos **inicialmente vacía**,
- **agregarle** obligación de cerrar un paréntesis (resp. corchete, llave) cada vez que se encuentra un paréntesis (resp. corchete, llave) que abre,
- **removerle** obligación de cerrar un paréntesis (resp. corchete, llave) (**comprobando** previamente que la constancia no sea vacía y que la **primera** obligación a cumplir sea justamente la de cerrar el paréntesis (resp. cochete, llave)) cada vez que se encuentra un paréntesis (resp. cochete, llave) que cierra,
- Al finalizar, **comprobar** que la constancia está vacía.

Pila

- Hace falta **algo**, una “constancia,” con lo que se pueda
  - inicializar vacía,
  - agregar una obligación de cerrar delimitador,
  - comprobar si quedan obligaciones,
  - examinar la primera obligación,
  - quitar una obligación.
- La última obligación que se agregó, es la primera que debe cumplirse y quitarse de la constancia.
- Esto se llama **pila**.

TAD Pila

- La pila se define por lo que sabemos: sus cinco operaciones
  - inicializar en vacía
  - apilar una nueva obligación (o elemento)
  - comprobar si está vacía
  - examinar la primera obligación (si no está vacía)
  - quitarla (si no está vacía).
- Nuevamente las operaciones **inicializar** y **agregar** son capaces de generar todas las pilas posibles,
- **comprobar** y **examinar**, en cambio, solamente examinan la pila,
- **quitarla** no genera más valores que los obtenibles por **inicializar** y **agregar**.

Especificacion del TAD Pila

module TADPila **where**

data Pila e = Vacía
    | Apilar e (Pila e)

es_vacía :: Pila e $\rightarrow$ Bool
primero :: Pila e $\rightarrow$ e
desapilar :: Pila e -> Pila e
- - las dos últimas se aplican sólo a pila no Vacía

es_vacía Vacía = True
es_vacía (Apilar e p) = False

primero (Apilar e p) = e
desapilar (Apilar e p) = p

Especificacion del TAD Pila

```
module TADPila where

data Pila e = Vacía
            | Apilar e (Pila e)

es_vacía :: Pila e → Bool
primero :: Pila e → e
desapilar :: Pila e -> Pila e
- - las dos últimas se aplican sólo a pila no Vacía

es_vacía Vacía = True
es_vacía (Apilar e p) = False

primero (Apilar e p) = e
desapilar (Apilar e p) = p
```

Explicación

Los valores posibles de una Pila están expresados por

- ningún elemento: Vacía
- un elemento: Apilar ')' Vacía, , Apilar ']' Vacía, Apilar '}' Vacía
- dos elementos: Apilar ')' (Apilar ')' Vacía) Apilar ')' (Apilar ']'Vacía) . . .
- tres elementos: Apilar ')' (Apilar ')' (Apilar ']' Vacía)) . . .
- cuatro elementos: Apilar ')' (Apilar ')' (Apilar ']' (Apilar '}' Vacía))) . . .
- etcétera

Sobre las ecuaciones de **es_vacía**

- ¿Cómo nos convencemos de que las ecuaciones de **es_inicial** cubren todos los casos posibles?
- Comenzamos escribiendo

  es_vacía p = ¿?

  donde p es una variable que representa una pila arbitrario.
- Pero no sabemos qué escribir en la parte derecha porque el resultado depende de la pila p.

Sobre las ecuaciones de **es_vacía**

- Reemplazamos la pila arbitraria p por cada uno de los casos posibles:

  es_vacía Vacía = ¿?
  es_vacía (Apilar e p) = ¿?

- Ahora sí estamos en condiciones de saber cuál debe ser el resultado en cada caso:

  es_vacía Vacía = True
  es_vacía (Apilar e p) = False

Sobre las ecuaciones de **primero**

- Comenzamos escribiendo

  primero p = ¿?

  donde p es una variable que representa una pila arbitraria **salvo Vacía.**
- La reemplazamos por cada uno de los casos **posibles**:

  primero (Apilar e p) = ¿?
- Ahora sí estamos podemos completar el resultado:

  primero (Apilar e p) = e
- De manera similar para **desapilar.**

Especificación y prototipo

Mostrar en Haskell.

Implementación

type stack = . . . {- no sabemos aún cómo se implementará -}

proc empty(**out** p:stack) {Post: p $\sim$ Vacía}

{Pre: p $\sim$ P $\wedge$ e $\sim$ E}
proc push(**in** e:elem,**in**/**out** p:stack)
{Post: p $\sim$ Apilar E P}

{Pre: p $\sim$ P $\wedge \neg$is_empty(p)}
fun top(p:stack) **ret** e:elem
{Post: e $\sim$ primero P}

Implementacion

{Pre: p $\sim$ P $\wedge \neg$is_empty(p)}
proc pop(**in**/**out** p:stack)
{Post: p $\sim$ desapilar P}

fun is_empty(p:stack) **ret** b:**bool**
{Post: b $=$ (p $\sim$ Vacía)}

Algoritmo de control de delimitadores balanceados

fun matching_delimiters (a: **array**[1..n] **of char**) **ret** b: **bool**
  var i: **nat**
  var p: stack **of char**
  b:= **true**
  empty(p)
  i:= 1
  do $i \leq n \wedge b \rightarrow$ **if** left(a[i]) $\rightarrow$ push(match(a[i]),p)
    right(a[i]) $\wedge$ (is_empty(p) $\vee$ top(p) $\neq$ a[i]) $\rightarrow$ b:= **false**
    right(a[i]) $\wedge$ $\neg$is_empty(p) $\wedge$ top(p) $=$ a[i] $\rightarrow$ pop(p)
    otherwise $\rightarrow$ **skip**
   fi
   i:= i+1
  od
  b:= b $\wedge$ is_empty(p)
end fun

Este algoritmo asume, además de la implementacion de pila,

- una función **match** tal que match('(') = ')', match('[') = ']', match('{') = '}', etc.
- una función **left**, tal que left('('), left('['), left('{'), etc son verdadero, en los restantes casos left devuelve falso.
- una función **right**, tal que right(')'), right(']'), right('}'), etc son verdadero, en los restantes casos, right devuelve falso.