

Algoritmos y Estructuras de Datos II

Tipos Abstractos de Datos (TADs o ADTs en inglés)

4 de abril de 2016

Clase de hoy

- 1 Repaso
 - Tipos concretos versus abstractos
 - Tipos abstractos de datos
- 2 Tipos abstractos de datos (TADs)
- 3 Paréntesis balanceados
 - TAD Contador
 - Especificación del TAD Contador
 - Sobre la especificación
 - Resolviendo el problema
- 4 Generalización de paréntesis balanceados
 - TAD Pila
 - Especificación del TAD Pila
 - Resolviendo el problema

Repaso

- cómo vs. qué
- 3 partes
 - 1 análisis de algoritmos
 - algoritmos de ordenación
 - operación elemental
 - conteo
 - propiedades y jerarquía
 - recurrencias divide y vencerás
 - 2 tipos de datos
 - tipos concretos: arreglos, listas, tuplas, punteros.
 - tipos abstractos
 - 3 técnicas de resolución de problemas

Tipos de datos

Conceptualmente distinguimos dos clases de tipos de datos:

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

Tipos abstractos de datos (TADs)

- Identificarlos es una tarea **siempre recompensada**.
 - mayor entendimiento cabal del problema a resolver,
 - algoritmos en un lenguaje comprensible por el ser humano,
 - permite poner a prueba (a través de prototipos) tempranamente si la solución está bien encaminada,
 - facilita el diseño, la organización del trabajo de programación y el trabajo en equipo,
 - proporciona una solución independiente de la representación,
 - da lugar a mejores soluciones: programas más elegantes, más legibles, más breves, más eficientes, más mantenibles, más fáciles de reutilizar, más factiblemente libre de errores.
- Ejemplificaremos a través de una serie de problemas, cada uno de ellos dará lugar a un tipo abstracto.

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 \rightarrow Bool

decrementar :: Contador \rightarrow 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 $\rightarrow 0$
- Incrementar Inicial $\rightarrow 1$
- Incrementar (Incrementar Inicial) $\rightarrow 2$
- Incrementar (Incrementar (Incrementar Inicial)) $\rightarrow 3$
- etcétera,
- es interesante observar que los números naturales parecen más adecuados que los enteros, a pesar de que nuestra primera solución era con enteros.

Intuitivamente

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

- Inicial \rightarrow
- Incrementar Inicial $\rightarrow |$
- Incrementar (Incrementar Inicial) $\rightarrow ||$
- Incrementar (Incrementar (Incrementar Inicial)) $\rightarrow |||$
- 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

Mostrarlo en Haskell.

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 precondition
- 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.

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.
- Mostrar en Haskell.

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 ~ Inicial}

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

{Pre: c ~ C \wedge \neg is_init(c)}

proc dec (**in/out** c: counter)

{Post: c ~ decrementar C}

fun is_init (c: counter) **ret** b: **bool** {Post: b = (c ~ 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  $i \leq n \wedge b \rightarrow$  if a[i] = '('  $\rightarrow$  inc(c)
    a[i] = ')'  $\wedge$  is_init(c)  $\rightarrow$  b:= false
    a[i] = ')'  $\wedge$   $\neg$ is_init(c)  $\rightarrow$  dec(c)
    otherwise  $\rightarrow$  skip
  fi
  i:= i+1
od
  b:= b  $\wedge$  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. corchete, llave)) cada vez que se encuentra un paréntesis (resp. corchete, 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**.

Especificación 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 \rightarrow 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

Especificación 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

Especificación y prototipo

Mostrar en Haskell.

Implementación

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

proc empty(**out** p:stack) {Post: p ~ Vacía}

{Pre: p ~ P \wedge e ~ E}

proc push(**in** e:elem,**in/out** p:stack)

{Post: p ~ Apilar E P}

{Pre: p ~ P \wedge \neg is_empty(p)}

fun top(p:stack) **ret** e:elem

{Post: e ~ primero P}

Implementación

{Pre: $p \sim P \wedge \neg \text{is_empty}(p)$ }

proc pop(**in/out** p:stack)

{Post: $p \sim \text{desapilar } P$ }

fun is_empty(p:stack) **ret** b:bool

{Post: $b = (p \sim \text{Vacía})$ }

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 implementación de pila,

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