Los Tipos Abstractos de Datos

Estructuras de Datos y Algoritmos

03/04

- ¿Qué son los TAD's?
- Metodología de programación con TAD's
- Especificación Algebraica de TAD's

¿Qué son los TAD's?

- Con los lenguajes de programación estructurados (años 60) surge el concepto de tipo de datos.
- Ese concepto es insuficiente para soft a gran escala: sólo el compilador restringe el uso de los datos.
- En los 70 aparece el concepto de TAD: un tipo de datos no sólo es el conjunto de valores, sino también sus operaciones con sus propiedades.

¿Pero qué son los TAD's?

- El concepto de TAD ya existe en los lenguajes de programación estructurados: **los tipos predefinidos**.

Ejemplo:

Definición del tipo de datos de los enteros en ADA

- valores: los del intervalo [INTEGER'FIRST,INTEGER'LAST]
- operaciones: +, -, *, /, resto, módulo, valor absoluto, exp.
- propiedades de las operaciones: a+b=b+a, ...

No hay que saber nada sobre implementación

Definición de TAD

- Un Tipo Abstracto de Datos es un conjunto de valores y de operaciones definidos mediante una **especificación independiente de cualquier representación**.

TAD = valores + operaciones

Definición de TAD

- La manipulación de un TAD sólo depende de su especificación, **nunca** de su implementación.

Para manipular los enteros nos olvidamos de cómo se representan los valores y de cómo están implementadas las operaciones.

Ejemplo de TAD

- Los números complejos con las operaciones de suma, producto, parte real y parte imaginaria

¿Cómo lo especificamos?

Especificación / implementación

- Dada una especificación de TAD hay muchas implementaciones válidas.
- Un cambio de implementación de un TAD es transparente a los programas que lo utilizan.

Trabajando con TAD's

- Se pueden implementar los TAD's sólo a partir de la especificación, sin saber para qué se van a usar.

  Saber qué sin necesidad de para qué.

  Reusabilidad

- Se pueden utilizar los TAD's sólo conociendo la especificación.

  Saber qué sin necesidad de cómo

  Seguridad

Trabajando con TAD's

- Se puede cambiar la implementación un TAD utilizado a otra más eficiente.
- Se pueden implementar los TAD's por separado y después integrarlos.

Legibilidad

Diseño modular

Organización del trabajo

Corrección

- Esto facilita el mantenimiento.

Modificabilidad

Trabajando con TAD's

- Se puede cambiar la implementación un TAD utilizado a otra más eficiente.
- Se pueden implementar los TAD's por separado y después integrarlos.

Legibilidad

Diseño modular

Organización del trabajo ← es mejor que el diseño descendente

Corrección

- Esto facilita el mantenimiento.

Modificabilidad

Trabajando con TAD's

- Se puede demostrar la corrección de una implementación a partir de su especificación.

Más útil, se pueden generar prototipos a partir de la especificación algebraica

Ejercicios

- Especificar (no formalmente) los siguientes TAD's
  - Cadenas de caracteres
  - Fechas

Programar con TAD's

Estructuras de Datos y Algoritmos

03/04

Definición de TAD

- Un Tipo Abstracto de Datos es un conjunto de valores y de operaciones definidos mediante una **especificación independiente de cualquier representación**.

TAD = valores + operaciones

Programar con TAD's

1. Especificación:

- Establecer la interfaz con el usuario del tipo ("lo que necesita saber el usuario")

  Decir qué es sin decir nada sobre cómo se hace

- Se trata de dar la lista de operaciones necesarias y especificarlas.
- Debe ser precisa, legible y no ambigua
- Nosotros usaremos especificación algebraica

Ejemplo de especificación (1)

espec booleanos

géneros bool

operaciones

verdad: $\rightarrow$ bool

falso: $\rightarrow$ bool

$\neg$_: bool $\rightarrow$ bool

$\wedge$,_$\vee$_: bool bool $\rightarrow$ bool

Ejemplo de especificación (1)

ecuaciones b: bool;

$\neg$verdad = falso

$\neg$falso = verdad

verdad $\vee$ falso = verdad

b $\vee$ verdad = verdad

b $\vee$ falso = b

b $\wedge$ verdad = b

b $\wedge$ falso = falso

fespec

Ejemplo de especificación (1)

espec booleanos

géneros bool

conjunto de valores

operaciones

verdad: $\rightarrow$ bool

falso: $\rightarrow$ bool

$\neg\_$: bool $\rightarrow$ bool

$\_\wedge\_, \_\vee\_$: bool bool $\rightarrow$ bool

Ejemplo de especificación (1)

espec booleanos

géneros bool

operaciones

verdad: $\rightarrow$ bool

falso: $\rightarrow$ bool

$\neg$_: bool $\rightarrow$ bool

$\wedge$,_$\vee$_: bool bool $\rightarrow$ bool

operaciones

Ejemplo de especificación (1)

ecuaciones b: bool;

¬verdad = falso

¬falso = verdad

verdad ∨ falso = verdad

b ∨ verdad = verdad

b ∨ falso = b

b ∧ verdad = b

b ∧ falso = falso

fespec

propiedades de las operaciones

Ejemplo de especificación (2)

espec conjuntos

usa booleanos,caracteres,naturales

género conjcar

operaciones

vacío: $\rightarrow$ conjcar

poner: carácter conjcar $\rightarrow$ conjcar

$\cup$: conjcar conjcar $\rightarrow$ conjcar

$\cap$: conjcar conjcar $\rightarrow$ conjcar

_$\in$_: carácter conjcar $\rightarrow$ booleano

...

Ejemplo de especificación (2)

ecuaciones A,B: conjcar;c,c1,c2:carácter

$A \cup$ vacío = A

$A \cup$ poner(c,B) = poner(c,$A \cup B$)

$A \cap$ vacío = vacío

$c \in A \rightarrow A \cap$ poner(c,B) = poner(c,$A \cap B$)

$c \notin A \rightarrow A \cap$ poner(c,B) = $A \cap B$

$c \in$ vacío = falso

...

fespec

Programar con TAD's

2. Implementación:

- Elegir la representación de los valores
- Implementar las operaciones

- Debe ser estructurada, legible y eficiente

Implementación

Una propiedad deseable es la **encapsulación**.

- Representación privada: el usuario no conoce los detalles de la implementación.
- Tipo protegido: El usuario sólo puede utilizar las operaciones previstas.

ADA es muy adecuado, tiene buena encapsulación.

```
package conjCaracteres is
 type conjcar is private;
 procedure vacio(A:out conjcar);
 function esVacio(A:in conjcar) return boolean;
 procedure poner(c:in character; A:in out conjcar);
 procedure quitar(c:in character; A:in out conjcar);
 function pertenece(c:in character; A:in conjcar) return
boolean;
 procedure union(A,B:in conjcar; C:out conjcar);
 procedure interseccion(A,B:in conjcar; C:out conjcar);
 function cardinal(A:in conjcar) return integer;
private
  ...
end conjCaracteres;
```

```
generic
  type ind is (<>); -- cualquier tipo discreto
  type elem is private; -- cualquier tipo
  with function ">"(a,b:elem) return boolean;

package ordenacion_g is
  type vector is array(ind range <>) of elem;
  procedure ordena(v:in out vector);
end ordenacion_g;
```

```
with ordenacion_g;
procedure titi is
 type color is (rojo,azul,gris);
 type dia is (lu,ma,mi,ju,vi,sa,dom);
 package miord is new ordenacion_g(dia,color,">");
 use miord;
 x:vector(ma..vi):=(gris,azul,rojo,gris);
begin
  ...
  ordena(x);
  ...
end titi;
```