Universidad Rey Juan Carlos Curso 1999/00

Ingenierías Técnicas en Informática de Sistemas y de Gestión

Estructuras de Datos y de la Información

Examen Parcial Febrero 2000 Fecha: 8−2−2000

Normas:

- La duración del examen es de 3 horas.
- Todos los ejercicios se entregarán en hojas separadas.
- En cada ejercicio se indica su puntuación total y el valor de los apartados que lo componen.

Ejercicio 1. (Puntuación total: 3 puntos)

Resolver los siguientes problemas correspondientes al uso de Tipos Abstractos de Datos (TADs):

- **(1 punto)** Sea una secuencia de elementos almacenados en una cola. Se desea comprobar si dicha secuencia, *en orden inverso*, es un prefijo de una pila de elementos del mismo tipo. Se considera que una cola vacía cumple dicha propiedad. Se pide especificar formalmente una función *Sombrero* que, haciendo uso de las operaciones de los TADs *TipoPila* y *TipoCola*, decida si se cumple la propiedad anterior. La sintaxis de dicha operación es:

Sombrero: TipoCola × TipoPila ! Booleano

Por ejemplo, dada la pila *P =5,8,3,2,4],* en la que *5* es la cima:

Sombrero ([3,8,5], P) = cierto Sombrero ([], P) = cierto

Sombrero ([3,8], P) = falso Sombrero ([4,8,5], P) = falso

- **(2 puntos)** Para trabajar con *árboles binarios de búsqueda* se necesita una operación *NumNodosMenores* que reciba como entradas un árbol binario de búsqueda y un elemento y devuelva el número de nodos del árbol que son estrictamente menores que el elemento dado. Por ejemplo, sea el árbol binario de búsqueda A (con elementos de tipo entero):

10

- 20

5 9 18 30

NumNodosMenores (A, 10) = 3 NumNodosMenores (A, 7) = 1

NumNodosMenores (A, 20) = 5 NumNodosMenores (A, 4) = 0

Se pide extender el TAD *TipoArbolBB* con la operación *NumNodosMenores*, dando para ello su especificación algebraica. Se recuerda que el TAD *TipoArbolBB* consta, entre otras, de las operaciones

generadoras *CrearABBVacio* y *ConstruirABB*, las operaciones selectoras *Raiz*, *HijoIzq* e *HijoDer* y la observadora *EsABBVacio*. Se supondrá también que está disponible la operación *NumNodos: TipoArbolBB ! Natural*, que devuelve el número total de nodos de un árbol binario de búsqueda.

Ejercicio 2. (Puntuación total: 4 puntos)

Una cadena de tiendas de material informático necesita un programa para gestionar el almacén en el que acumula sus productos. Los productos se identifican de forma unívoca mediante un nombre, y se necesita conocer en todo momento la cantidad de unidades de cada producto que están disponibles en el almacén. En el diseño del programa se han identificado dos Tipos Abstractos de Datos: el TAD *TipoProducto* y el TAD *TipoAlmacen*. Se pide:

- **(0.5 puntos)** Especificar algebraicamente el TAD *TipoProducto*. Éste, además de las operaciones constructoras generadoras y observadoras selectoras que se consideren necesarias, deberá incluir las dos operaciones siguientes:

* *EliminarUnidad: TipoProducto ! TipoProducto* (resta uno al número de unidades del producto, salvo que este número sea nulo).

* *AñadirUnidades: TipoProducto × Positivo ! TipoProducto* (añade al producto la cantidad de unidades indicada en segundo lugar).

La especificación deberá constar de los apartados habituales (véase como ejemplo la especificación del TAD *TipoLista* adjunta).

- **(1 punto)** El TAD *TipoAlmacen* incluye, entre otras, las siguientes operaciones:

    - Constructoras generadoras:

* *CrearAlmacenVacio: ! TipoAlmacen*

* *AñadirProducto*: *TipoAlmacen × TipoProducto ! TipoAlmcen*

    - Otras operaciones:

* *EsAlmacenVacio: TipoAlmacen ! Booleano* (decide si un almacén está vacío).

* *ProductoEstrella: TipoAlmacen ! TipoProducto* (si el almacén tiene productos, devuelve el que tiene más unidades disponibles).

* *ServirProducto: TipoAlmacen × String ! TipoAlmacen (si existe en el almacén un producto cuyo nombre coincide con la cadena de entrada, y éste tiene unidades disponibles, la operación registra la salida de una unidad de dicho producto).*

Para cada una de las *tres* últimas operaciones, se pide: 1) su categoría (constructora no generadora, observadora selectora o no selectora, constructora no generadora), 2) su tipo de dominio −parcial o total−, 3) las ecuaciones de definitud, en caso necesario y 4) la o las ecuaciones que definen su semántica, teniendo en cuenta que la expresión *AñadirProducto(almacen,producto)* garantiza que *producto* no está en *almacen*.

- **(0.75 puntos)** Para cada uno de los dos TADs identificados, justificar la elección de la estructura de datos que se considere más adecuada para su representación (se suponen disponibles, además de las estructuras estándar, todas las estructuras de datos estudiadas en clase). Escribir los ficheros de *Ada 95* necesarios para

describir la interfaz de cada uno de los dos TADs (ficheros *.ads*), *excluyendo* en ambos casos las cabeceras de todas sus operaciones.
- **(1.75 puntos)** De acuerdo con la representación elegida en el apartado anterior, implementar en *Ada 95* las operaciones *ProductoEstrella* y *ServirProducto* del TAD *TipoAlmacen* (parte correspondiente al fichero *.adb*), teniendo en cuenta los posibles casos excepcionales.

Ejercicio 3. (Puntuación total: 3 puntos)

Dado el TAD *TipoLista* y su implementación mediante una secuencia de nodos simplemente enlazados (véase en hoja adjunta la especificación algebraica y la interfaz de dicho TAD), se pide:

- **(1 punto)** Añadir a la especificación algebraica del TAD *TipoLista* la operación constructora *AgruparElemento*, especificando sus ecuaciones. La operación tiene la siguiente sintaxis:

AgruparElemento: TipoElemento x TipoLista ! TipoLista

Dicha operación agrupa de forma consecutiva todas las apariciones del elemento en la lista, a partir de la primera ocurrencia. Las posiciones relativas del resto de los elementos no varían. Por ejemplo:

AgruparElemento(1,[1,3,4,1,4]) " [1,1,3,4,4]

AgruparElemento(1,[3,1,4,1,4,1,1]) " [3,1,1,1,1,4,4]

AgruparElemento(1,[3,4,1,4]) " [3,4,1,4]

AgruparElemento(2,[1,3,4]) " [1,3,4]

AgruparElemento(1,[]) " []

- **(1.5 puntos)** Implementar dicha operación en *Ada 95*, definiendo las modificaciones que habría que realizar en la interfaz (*lista.ads)* y en el cuerpo del paquete *(lista.adb*).
- **(0.5 puntos)** Estimar la complejidad de la operación *AgruparElemento*, programada en el apartado anterior, expresando el resultado en la notación asintótica O( ) y razonando la respuesta.

Universidad Rey Juan Carlos Curso 1999/00

Ingenierías Técnicas en Informática de Sistemas y de Gestión

Estructuras de Datos y de la Información

Soluciones Examen Parcial Febrero 2000

Ejercicio 1.

- La función de la que se pide su especificación formal recibe dos TADs, *TipoPila* y *TipoCola*, por lo que hay que escribir ecuaciones que contemplen el comportamiento de la función para cada generador. La especificación queda:

Sombrero: TipoCola × TipoPila ! Booleano

Variables:

pila : TipoPila;

cola : TipoCola;

elemento, e : TipoElemento;

Ecuaciones:

Sombrero (CrearColaVacia, CrearPilaVacia) " Cierto

Sombrero (CrearColaVacia, Apilar (e, pila)) " Cierto

Sombrero (Insertar (elemento, cola), CrearPilaVacia) " Falso

Sombrero (Insertar (elemento,cola), Apilar (e,pila)) "

SI elemento /= e ! Falso

| Sombrero (cola, pila)

(Nota: las dos primeras ecuaciones se pueden agrupar en : Sombrero (CrearColaVacia, pila) " Cierto)

- En este ejercicio se pide *exclusivamente* completar la especificación algebraica de la operación NumNodosMenores que se quiere añadir al TAD TipoArbolBB (árbol binario de búsqueda). Esta operación tiene la siguiente sintaxis:

NumNodosMenores: TipoArbolBB × TipoElemento ! Natural

Para escribir su semántica basta especificar el comportamiento de esta operación en función de las operaciones constructoras generadoras del TAD TipoArbolBB (CrearABBVacio y ConstruirABB). Como dice el enunciado del problema, se dispone de una función auxiliar NumNodos que devuelve el número de nodos de un árbol binario de búsqueda. Con todo ello, la extensión del TAD queda:

NumNodosMenores (CrearABBVacio, dato) " 0

NumNodosMenores (ConstruirABB (ai, raiz, ad), dato) "

SI dato = raiz ! NumNodos (ai)

| SI dato < raiz ! NumNodosMenores (ai, dato)

| 1 + NumNodos (ai) + NumNodosMenores (ad, dato)

Ejercicio 2.

- Para especificar algebraicamente el TAD *TipoProducto* hay que tener en cuenta las siguientes consideraciones:

    - operaciones constructoras generadoras: todo TAD tiene que tener necesariamente como operaciones constructoras generadoras aquellas, y sólo aquellas, que permitan construir *cualquier* valor del tipo. Un producto está definido exclusivamente por un nombre y un número que representa la cantidad de unidades del producto disponibles, luego para construir cualquier producto bastará con una única

operación constructora generadora que reciba como entrada los dos datos citados y construya a partir de ellos un valor del tipo *TipoProducto*.

- operaciones observadores selectoras: son aquellas que permiten acceder a las distintas partes de las que consta un elemento del tipo de datos que se está definiendo. En este caso serán por lo tanto necesarias dos operaciones selectoras que devuelvan los dos atributos que han sido utilizados para construir un producto (su nombre y la cantidad de unidades disponibles).
- operaciones constructoras no generadoras: las dos operaciones facilitadas en el enunciado son operaciones constructoras −devuelven un valor del tipo− , pero no son generadoras puesto que no son imprescindibles para construir productos: simplemente permiten modificar uno de sus atributos.

TAD TipoProducto

OPERACIONES

* Constructoras generadoras

CrearProducto: String × Natural ! TipoProducto

* Observadoras selectoras

DameNombre: TipoProducto ! String

DameCantidad: TipoProducto ! Natural

* Constructoras no generadoras

EliminarUnidad: TipoProducto ! TipoProducto

AñadirUnidades: TipoProducto × Positivo ! TipoProducto

VARIABLES

nombre: String; cantidad: Natural; nuevas: Positivo;

ECUACIONES

DameNombre (CrearProducto (nombre, cantidad)) " nombre

DameCantidad (CrearProducto (nombre, cantidad)) " cantidad

EliminarUnidad (CrearProducto (nombre, cantidad)) "

SI cantidad = 0

! CrearProducto (nombre, cantidad)

- CrearProducto (nombre, cantidad − 1)

AñadirUnidades (CrearProducto (nombre, cantidad), nuevas) "

CrearProducto (nombre, cantidad + nuevas)

FTAD

- El enunciado muestra que el TAD *TipoAlmacen* USA el TAD *TipoProducto*, por lo que todas las operaciones definidas en este último son accesibles desde el primero. A la hora de definir la semántica de las operaciones es además importante tener en cuenta −como se dice en el enunciado− que la operación generadora *AñadirProducto* garantiza que cada producto distinto aparece una única vez en el almacén.

    - *EsAlmacenVacio* es una operación observadora −devuelve un valor que no pertenece al tipo−, no selectora −lo que devuelve no representa una parte del tipo− y es total puesto que está definida para cualquier entrada. Su semántica es:

EsAlmacenVacio (CrearAlmacenVacio) " cierto

EsAlmacenVacio(AñadirProducto(almacen, producto)) " falso

    - *ProductoEstrella* es una operación observadora no selectora. Es parcial puesto que cuando el almacén está vacío es imposible devolver ningún valor del tipo *TipoProducto*. Su ecuación de definitud es por lo tanto:

DEF (ProductoEstrella (AñadirProducto(almacen, producto)))

y su semántica viene dada por la siguiente ecuación:

ProductoEstrella (AñadirProducto(almacen, producto)) "

SI EsAlmacenVacio (almacen) "

DameCantidad(producto) > DameCantidad (ProductoEstrella(almacen))

! producto

| ProductoEstrella(almacen)

    - *ServirProducto* es una operación constructora no generadora −permite modificar valores del tipo pero no es imprescindible para generarlos− que puede considerarse o bien total (servir un producto en un almacén vacío consiste en no hacer nada) o bien parcial (la operación no se define cuando el almacén está vacío). Si se contempla el primer caso −el enunciado no exige tener en cuenta el segundo− la operación no tiene ecuaciones de definitud, y las ecuaciones que definen su semántica son dos:

ServirProducto (CrearAlmacenVacio, nombre) " CrearAlmacenVacio

ServirProducto (AñadirProducto(almacen, producto), nombre) "

SI nombre = DameNombre (producto)

! AñadirProducto (almacen, EliminarUnidad(producto))

| AñadirProducto (ServirProducto(almacen, nombre), producto)

-
    - **TAD TipoProducto**

Un producto está compuesto únicamente por dos valores (un nombre y un número de unidades), por lo que la forma adecuada de implementar el TAD *TipoProducto* es mediante un registro que contenga esos dos datos como campos.

Fichero *ads* (salvo declaración de operaciones)

```
PACKAGE Producto IS

−− constantes y tipos de datos

MAX_NOMBRE: CONSTANT Positive := 20;

SUBTYPE TipoNombre IS String(1..MAX_NOMBRE);

TYPE TipoProducto IS PRIVATE;

.... (declaración de operaciones)

PRIVATE

TYPE TipoProducto IS

RECORD

nombre: TipoNombre;

cantidad: Natural;

END RECORD;

END Producto;
```

- **TAD TipoAlmacén**

Un almacén es una colección de productos (elementos del TAD *TipoProducto*), y por lo tanto la forma más adecuada de implementarlo es mediante el TAD *TipoLista* estudiado en clase: bastará con instanciar este TAD genérico con *TipoProducto* como parámetro. No tiene sentido implementarlo directamente con punteros y variables dinámicas puesto que se dispone del TAD genérico *TipoLista* con todas las operaciones necesarias para manipular cualquier lista de elementos.

Fichero *ads* (salvo declaración de operaciones)

```
WITH Lista;

WITH Producto;

PACKAGE Almacen IS

−− tipo de datos (necesariamente privado limitado puesto que se

−− implementa mediante un tipo de datos privado limitado)
```

TYPE TipoAlmacen IS LIMITED PRIVATE;

−− excepciones (necesaria para la operación Parcial *ProductoEstrella*)

AlmacenVacio: EXCEPTION;

.... (declaración de operaciones)

PRIVATE

−− Instanciación del TAD Lista

PACKAGE LProductos IS NEW Lista (Producto.TipoProducto);

−− Definición del tipo almacén

TYPE TipoAlmacen IS

RECORD

productos: LProductos.TipoLista;

END RECORD;

END Almacen;

- A continuación se incluye la implementación iterativa de las dos operaciones pedidas (ambas podrían implementarse también de forma recursiva). En ambos casos es fundamental tener presente que los dos tipos de datos que intervienen en la implementación del tipo *TipoAlmacen*, *TipoProducto* y *TipoLista*, son Tipos Abstractos de Datos y por lo tanto la *única* forma posible de manipularlos es mediante las operaciones ofrecidas por los paquetes correspondientes (en el caso de *TipoProducto,* al haberse definido como un tipo privado pero no limitado, también es posible usar las operaciones estándar del lenguaje *Ada 95* de asignación e igualdad/desigualdad).

    - **Implementación de la función *ProductoEstrella***

FUNCTION ProductoEstrella (almacen: TipoAlmacen)

RETURN Producto.TipoProducto IS

−− COMPLEJIDAD: 0(n), siendo n el número de productos en el almacén.

producto_estrella: Producto.TipoProducto;

copia: LProductos.TipoLista;

BEGIN

producto_estrella := LProductos.Primero(almacen.productos);

−− Si el almacén está vacío la operación Primero del TAD Lista

−− genera la excepción 'ListaVacia', que se captura más adelante.

LProductos.Copiar (copia, LProductos.Resto(almacen.productos));

WHILE NOT LProductos.EsVacia (copia) LOOP

IF Producto.DameCantidad (LProductos.Primero(copia)) >

Producto.DameCantidad (producto_estrella) THEN

producto_estrella := LProductos.Primero(copia);

END IF;

LProductos.Copiar (copia, LProductos.Resto(copia));

END LOOP;

RETURN producto_estrella;

EXCEPTION

WHEN LProductos.ListaVacia

=> Ada.Exceptions.Raise_Exception

(AlmacenVacio'identity, El almacén está vacío);

END ProductoEstrella;

- **Implementación del procedimiento *ServirProducto***

PROCEDURE ServirProducto (almacen: IN OUT TipoAlmacen;

nombre: IN Producto.TipoNombre) IS

−− COMPLEJIDAD: O(n), siendo n el número de productos en el almacén

copia: LProductos.TipoLista;

prod: Producto.TipoProducto;

BEGIN

LProductos.Copiar (copia, almacen.productos);

WHILE NOT LProductos.EsVacia (copia) AND THEN

Producto.DameNombre(LProductos.Primero(copia)) /= nombre LOOP

LProductos.Copiar (copia, LProductos.Resto(copia));

END LOOP;

IF NOT LProductos.EsVacia (copia) THEN

prod := LProductos.Primero (copia);

LProductos.BorrarElemento (prod, almacen.productos);

Producto.EliminarUnidad (prod);

LProductos.Construir (prod, almacen.productos);

LProductos.Destruir (copia);

END IF;

END ServirProducto;

Ejercicio 3.

- *TAD Lista*

.....

OPERACIONES

.....

AgruparElemento: TipoElemento x TipoLista ! TipoLista

.....

ECUACIONES

.....

AgruparElemento(e, CrearVacia) " CrearVacia

AgruparElemento(e, Construir(e1,l)) "

Si e1=e

! Si Pertenece(e,l)

! Construir(e, AgruparElemento(e,Construir(e,BorrarElemento(e,l))))

| Construir(e,l)

| Construir(e1,AgruparElemento(e,l))

FIN TAD

Obsérvese que la operación *BorrarElemento(e,l)* sólo borra la primera ocurrencia del elemento en la lista. De esta forma, mediante la expresión *Construir(e,BorrarElemento(e,l))* se mueve el elemento *e* a la cabeza de la lista.

Dado que la función no es parcial, no es necesario especificar sus ecuaciones de definitud.

- <<*Fichero* lista.ads>>

PACKAGE Lista IS

.....

PROCEDURE AgruparElemento(e: IN TipoElemento; l: IN OUT TipoLista);

−− POST: La lista contiene las distintas ocurrencias del elemento e

−− agrupadas a partir de la primera aparicición

.....

PRIVATE

.....

END Lista;

Cualquiera de las siguientes alternativas de implementación se considera igualmente correcta:

− recursiva, utilizando el resto de operaciones del TAD

− iterativa, utilizando el resto de operaciones del TAD

− iterativa, utilizando la representación de *TipoLista* basada en punteros.

A continuación se muestra la solución en base a la última de las tres alternativas.

<<*Fichero* lista.adb>>

PROCEDURE AgruparElemento(e: IN TipoElemento; l: IN OUT TipoLista) IS

primero, ant, act: TipoLista;

−− La variable `primero' apunta a la primera ocurrencia del elemento

−− en la lista; `act' apunta al nodo que se está tratando

−− actualmente, y `ant' al nodo tratado en el paso anterior.

BEGIN

ant := l;

```
act := l;
WHILE act /= NULL LOOP
-- Los casos a tener en cuenta son los siguientes:
-- 1) Nos encontramos la primera ocurrencia:
-- Actualizamos la variable `primero' y pasamos al siguiente.
-- 2) Nos encontramos una ocurrencia del elemento, que no es la
-- primera, y además no se encuentra agrupada con las
-- anteriores: se agrupa y se pasa al siguiente nodo.
-- 3) Otro caso distinto de los anteriores: se pasa al siguiente.
IF primero=NULL AND act.elemento=e THEN
primero:=act;
-- Pasamos al siguiente elemento
ant := act;
act := act.siguiente;
ELSIF act.elemento=e AND ant.elemento /= e THEN
ant.siguiente := act.siguiente;
act.siguiente := primero.siguiente;
primero.siguiente := act;
-- Pasamos al siguiente elemento
act := ant.siguiente;
ELSE
-- Pasamos al siguiente elemento
ant := act;
act := act.siguiente;
END IF;
END LOOP;
```

END AgruparElemento;

- La complejidad del algoritmo iterativo, dado como implementación en el apartado anterior, se puede estimar de manera informal de la siguiente forma:

Para agrupar las distintas ocurrencias del elemento es necesario recorrer todos los elementos de la lista. Como el tratamiento que se realiza para cada elemento es de orden de complejidad constante, la complejidad de la operación es O(n), donde n=nº de elementos de la lista.

1