Práctica 6. TAD bolsa

Objetivos.

Se trata de construir el TAD BOLSA con una implementación no acotada con cabecera y una biblioteca que incluya las operaciones básicas sobre bolsas.

Se revisa en detalle el tratamiento de errores que debe realizarse en las bibliotecas y en el resto de clientes del TAD.

Enunciado.

Construir el TAD BOLSA según el siguiente módulo de definición:

```
DEFINITION MODULE Bolsa;
        FROM TadItem IMPORT ITEM;

        TYPE   BOLSA;
               TIPO_ERROR= <<a definir>>;

        PROCEDURE Error(): TIPO_ERROR;

        PROCEDURE Crear(): BOLSA;

        PROCEDURE EsVacia(b: BOLSA): BOOLEAN;
        PROCEDURE Esta(b: BOLSA; x: ITEM): BOOLEAN;
        PROCEDURE Cardinal(b:BOLSA): CARDINAL;
        PROCEDURE Pluralidad(b: BOLSA; x: ITEM): CARDINAL;

        PROCEDURE Incluir(VAR b: BOLSA; x: ITEM);
        (* Pre: Pluralidad(b, x) > 0 *)
        PROCEDURE Excluir(VAR b: BOLSA; x: ITEM);

        PROCEDURE Inicializar(b: BOLSA);
        PROCEDURE Elemento (b: BOLSA):ITEM;

        PROCEDURE Copiar(VAR b1:BOLSA; b2:BOLSA);

        PROCEDURE Destruir(VAR b: BOLSA);
END Bolsa.
```

Una *bolsa* (también llamada multiconjunto) es una colección de elementos similar a un conjunto, con la salvedad de que los elementos pueden aparecer repetidos. Al número de veces que se repite un elemento se le llama *pluralidad*. Por ejemplo:

{ A, F, K, A, A, F, T, A }

es una bolsa de caracteres donde la A tiene pluralidad 4 (aparece 4 veces), la F tiene pluralidad 2, y la K y la T tienen pluralidad 1.

El TAD BOLSA se representará mediante la siguiente estructura de datos:

```
TYPE   PNODO = POINTER TO NODO;

       NODO = RECORD
                        cont: ITEM;
                        pluralid: CARDINAL;
                        sig: PNODO;
              END;
```

```
BOLSA    = POINTER TO CABECERA;

CABECERA =  RECORD
                    primero: PNODO;
                    iter_elem: PNODO;
                    iter_plura: CARDINAL;
                END;
```

El procedimiento Excluir sólo elimina una de las apariciones del elemento. El Cardinal de una bolsa se define como la suma de las pluralidades de todos sus elementos. Los procedimientos Inicializar y Elemento implementan un iterador activo que recorre todos los elementos de la bolsa *dependiendo* de su pluralidad. Así, por ejemplo, si una bolsa está formada por los elementos 'R' de pluralidad 3 y 'K' de pluralidad 4, el procedimiento Elemento devolverá para las 3 primeras llamadas el elemento 'R' y para las 4 siguientes el elemento 'K'. Si se llama a Elemento tras haber devuelto el último elemento de la bolsa, se produce un error de Iteración Completa. Para poder empezar otro recorrido se ha de usar la operación de Inicializar que hace que la siguiente llamada a Elemento devuelva el primer elemento del recorrido. Para codificar el estado de este iterador activo, es necesario emplear un puntero (iter_elem) y un cardinal (iter_plura).

Construir una biblioteca sobre bolsas con las siguientes operaciones:

```
DEFINITION MODULE BibBolsa;
    FROM Bolsa IMPORT BOLSA;

    PROCEDURE EsIgual(b1, b2: BOLSA):BOOLEAN;
    PROCEDURE Union(VAR b:BOLSA;b1,b2: BOLSA);
    PROCEDURE Inter(VAR b:BOLSA;b1,b2: BOLSA);
    PROCEDURE Difer(VAR b:BOLSA;b1,b2: BOLSA);
    PROCEDURE DifSim(VAR b:BOLSA;b1,b2: BOLSA);

END BibBolsa.
```

El significado de las operaciones anteriores se ilustra en el siguiente ejemplo:

Unión: {2,3,5,7} $\cup$ {2,3,2,5} = {2,3,5,7,2,3,2,5}
Intersección: {2,3,5,7} $\cap$ {2,3,2,5} = {2,3,5}
Diferencia: {2,3,5,7} - {2,3,2,5} = {7}
Diferencia simétrica: Unión - Intersección, {2,3,5,7,2,3,2,5} - {2,3,5} = {7,2,3,2,5}

Finalmente, como aplicación hacer un programa que descomponga un número en factores primos y almacene estos factores en una bolsa. Desarrollar además los procedimientos Máximo Común Divisor (MCD(x,y)) y mínimo común múltiplo (mcm(x,y)). Para ello, tener en cuenta que el MCD(x,y) es igual al producto de los factores primos *comunes* a x e y, y que el mcm es igual al producto de los factores primos *no comunes* a x e y.

Tratamiento de Errores en Bibliotecas

Las bibliotecas que desarrollaremos a lo largo del curso no introducen tipos ni variables de error propias para el tratamiento de errores. Por el contrario, las bibliotecas se limitan a emplear el tipo TIPO_ERROR y la función Error exportadas por el TAD, como cualquier otro código cliente del TAD.

Cada vez que una operación de biblioteca invoque una operación del TAD, debe comprobar mediante la función Error si esta operación se ejecutó con éxito:

- Si la operación sobre el TAD se ejecutó con éxito, la operación de la biblioteca puede continuar en curso.
- Si la operación sobre el TAD terminó con error, el procedimiento de biblioteca debe dar por terminada su propia ejecución (sin emplear para ello un RETURN), cuidando de **no ejecutar ninguna operación adicional** sobre el TAD. De esta manera, nos aseguramos que la función Error del TAD siga devolviendo el error que se produjo originalmente.

Por ejemplo, aplicando la política de tratamiento de errores anteriormente descrita, el procedimiento Sumar de la biblioteca BibRac del TAD RACIONAL (práctica 1) queda como sigue:

```
PROCEDURE Sumar(VAR r:RACIONAL; r1,r2:RACIONAL);
VAR a,b,c,d:INTEGER;
BEGIN
        a:= Numerador(r1);
        IF Error() = SinError THEN
                b:= Denominador(r1);
                IF Error() = SinError THEN
                        c:= Numerador(r2);
                        IF Error() = SinError THEN
                                d:= Denominador(r2);
                                IF Error() = SinError THEN
                                        Asignar(r,a*d+c*b,b*d);
                                        (* aquí no hace falta comprobar el error *)
                                END;
                        END;
                END;
        END;
END Sumar;
```

El usuario de la biblioteca puede saber si el procedimiento de la biblioteca se ejecutó adecuadamente consultando el valor devuelto por la función Error del TAD, como si se tratara de una operación sobre el TAD:

```
Sumar(s,x,y);
IF Error()<>SinError THEN
        (* código si hay error *)
        VerError(Error());
        ...
ELSE
        (* código si no hay error *)
        ...
END;
```

Es decir, que desde el punto de vista del cliente, se realiza el **mismo tratamiento** para las operaciones del TAD y para las operaciones de la biblioteca.

El código del procedimiento Sumar pone de manifiesto un problema de nuestra técnica de tratamiento de errores: la inclusión explícita de comprobaciones de error tras cada operación del TAD resta legibilidad al código. La estructura lógica de la operación de suma de racionales queda oculta tras una cascada de condicionales que nada tiene que ver con la suma de racionales. En otras palabras, el código de la suma de racionales y el código del tratamiento de errores aparecen entrelazados. Desafortunadamente,

Modula-2 carece de los mecanismos adecuados para facilitar un tratamiento de errores *no invasivo* que permita separar el código en dos secciones: una para la operación propiamente dicha y otra para el tratamiento de errores.

Una posibilidad para aumentar la legibilidad del código consiste en simplificar el tratamiento de errores eliminando aquellas comprobaciones de error explícitas que resulten innecesarias. Se puede sustituir una llamada a la función Error y el código condicional asociado por **un comentario que justifique la eliminación de tal comprobación**. Por ejemplo, el procedimiento Sumar se puede simplificar como sigue:

```
PROCEDURE Sumar(VAR r:RACIONAL; r1,r2:RACIONAL);
VAR a,b,c,d:INTEGER;
BEGIN
        a:= Numerador(r1);
        IF Error() = SinError THEN
                c:= Numerador(r2);
                IF Error() = SinError THEN
                        b:= Denominador(r1);
                        (* r1 existe, pues Numerador(r1) no falló *)
                        d:= Denominador(r2);
                        (* r2 existe, pues Numerador(r2) no falló *)
                        Asignar(r,a*d+c*b,b*d);
                        (* aquí no hace falta comprobar el error *)
                END;
        END;
END Sumar;
```

No es posible que las llamadas a Denominador(r1) y Denominador(r2) provoquen error alguno, pues cuando se ejecutan ya es seguro que r1 y r2 han sido creados, y el único error que puede producirse en estas operaciones es ObjetoNoCreado. Esta técnica para simplificar el tratamiento de errores no sólo se aplica a la implementación de bibliotecas, también puede aplicarse a cualquier código cliente del TAD. Obviamente, para poder aplicar esta técnica con propiedad, es necesario que el usuario del TAD conozca qué tipo de errores se pueden presentar cada operación del TAD.

Práctica suplementaria

Se trata de implementar el algoritmo de control de una máquina expendedora.

Esta máquina dispone de varios artículos, que pueden estar repetidos o agotados, cada uno con su precio. El usuario introduce el importe, que no tiene que ser exacto, en una o varias monedas y selecciona el producto. En la máquina también se tienen las monedas recaudadas durante el día.

Si el importe es insuficiente o el producto está agotado, la máquina devuelve las monedas introducidas por el usuario. Si el importe es exacto, simplemente se sirve el producto. Si no, se calcula el cambio que se ha de devolver. Si no se puede devolver el cambio exacto, también se le devuelve las monedas al usuario. En caso contrario, se sirve el producto al cliente y se le devuelve el cambio.

A la hora de devolver el cambio intentará reducir el número de monedas (devolviendo el mayor número posible de monedas de valor alto).