Práctica 1. TAD número racional

Objetivos.

Con esta práctica se inicia al alumno en la *ocultación de información* y el concepto de módulo como código reutilizable. Se ha escogido un ejemplo sencillo antes de trabajar con tipos abstractos clásicos como son pila, cola, etc. Se hará un especial hincapié en la implementación de tipos abstractos de datos, mediante encapsulamiento de la información y la compilación separada.

En esta práctica se implementa el tipo RACIONAL con sus módulos de definición y de implementación. Haciendo importación de este tipo se construye una biblioteca con las operaciones para manipular los números racionales y haciendo importación de ambos módulos se construye un programa de prueba que compruebe su funcionamiento. Asimismo, se introduce el tratamiento de errores que se va a hacer en todos los programas a lo largo del curso.

Enunciado.

Se trata de construir un tipo abstracto del número racional, para ello el tipo RACIONAL será un tipo opaco definido en el módulo de implementación con la siguiente estructura:

```
TYPE NODO =  RECORD
                        num, den: INTEGER
             END;
RACIONAL = POINTER TO NODO
```

El módulo de definición es:

```
DEFINITION MODULE Racional;

    TYPE  RACIONAL;

    (*Pre: d <> 0 *)
    PROCEDURE Crear(n,d:INTEGER): RACIONAL;

    (* Pre: r ha sido creado. *)
    PROCEDURE Destruir(VAR r:RACIONAL);

    (* Pre: r ha sido creado. d <> 0 *)
    PROCEDURE Asignar(VAR r:RACIONAL;n,d:INTEGER);

    (* Pre: r ha sido creado. *)
    PROCEDURE Numerador(r:RACIONAL):INTEGER;

    (* Pre: r ha sido creado. *)
    PROCEDURE Denominador(r:RACIONAL):INTEGER;
END Racional.
```

El procedimiento Crear se encarga de reservar la memoria e inicializar el racional a n / d. Los racionales se almacenarán *normalizados*; es decir, el racional debe estar simplificado (dividiendo n y d por su máximo común divisor) y el signo debe afectar sólo al numerador. Por ejemplo, el racional 30 / -25 se almacenará como –6 / 5. Además el cero se representará siempre mediante su forma canónica 0 / 1. Este proceso de normalización debe llevarse a cabo en los procedimientos Crear y Asignar.

Haciendo uso de este TAD crear una biblioteca con las siguientes operaciones:

```
DEFINITION MODULE BibRac;
        FROM Racional IMPORT RACIONAL;

        (* Pre: r1 y r2 han sido creados. *)
        PROCEDURE Copiar(VAR r1: RACIONAL; r2:RACIONAL);

        (* Pre: r, r1 y r2 han sido creados. *)
        PROCEDURE Sumar(VAR r:RACIONAL;r1,r2:RACIONAL);

        (* Pre: r, r1 y r2 han sido creados. *)
        PROCEDURE Restar(VAR r:RACIONAL;r1,r2:RACIONAL);

        (* Pre: r, r1 y r2 han sido creados. *)
        PROCEDURE Multiplicar(VAR r:RACIONAL;r1,r2:RACIONAL);

        (* Pre: r, r1 y r2 han sido creados y r2 <> 0 *)
        PROCEDURE Dividir(VAR r:RACIONAL;r1,r2:RACIONAL);

        (* Pre: r1 y r2 han sido creados. *)
        PROCEDURE EsIgual(r1,r2:RACIONAL):BOOLEAN;
END BibRac.
```

Por último construir un programa de prueba que opere con el número racional, importando ambos módulos.

Tratamiento de errores.

En general, los procedimientos que se desarrollan están sujetos a unas condiciones para su correcto funcionamiento. Si hacemos una función que calcule la raíz cuadrada de un número, el parámetro no puede ser un número negativo o, en el caso de esta práctica, en el procedimiento Asignar, el denominador no puede valer 0. Este conjunto de condiciones se denomina *precondiciones* del procedimiento y suelen ser función de los parámetros. Si al invocar al procedimiento no se cumplen sus precondiciones, el resultado del procedimiento es incorrecto, se produce un error.

En primer lugar, los tipos abstractos que vamos a diseñar deben ofrecer las herramientas suficientes para que el usuario sepa si la precondición se cumple o no a la hora de llamar al procedimiento. Por ejemplo, en la definición del tipo PILA, con la operación Desapilar, que elimina el elemento que está en la cima de la pila, es razonable poner como precondición de la operación que la pila no esté vacía. En ese caso, el tipo Pila también debe disponer de una operación (por ejemplo PilaVacia), que nos diga si la pila tiene elementos o no. El uso previo de esta operación nos permitirá usar de forma segura la operación Desapilar.

Desafortunadamente, no siempre es tan fácil, o incluso posible, la definición de estas operaciones "de seguridad". Además, no hay forma de asegurarse de que el usuario las utilice adecuadamente en su programa. Es, por tanto, necesario el tratamiento de estas situaciones en las que no se cumple la precondición y se produce un error. Hay diferentes formas de controlar errores.

Ignorar errores. La forma más sencilla para el diseñador del tipo abstracto es descargar toda la responsabilidad en el usuario del tipo y suponer que siempre se cumplen las precondiciones. Si no es ese el caso y, por ejemplo, se llama a la operación Desapilar sobre una pila vacía, el programa tendrá un comportamiento impredecible. Esta política va

en contra de la robustez del programa (comportamiento correcto frente a situaciones de error o inesperadas) y, normalmente, no es aceptable.

Control silencioso. Otra forma, un poco más considerada con el usuario, es comprobar si se cumplen las precondiciones y, en caso contrario, no hacer nada. Es decir, si el procedimiento detecta que la pila sobre la que se quiere Desapilar está vacía, simplemente acaba su ejecución. Esta estrategia es peligrosa porque el usuario del tipo puede seguir trabajando pensando que la operación se ha ejecutado satisfactoriamente (pues no tiene información de lo contrario) y llegar a continuar el programa de forma errónea.

Abortar el programa. En el otro extremo de las soluciones tenemos otra, que consiste en emitir un mensaje por pantalla indicando el tipo de error detectado y salir del procedimiento mediante la función HALT.

Por ejemplo:

```
PROCEDURE Desapilar (VAR p: PILA);
    ...
        IF esVacia(P) THEN
            WrStr('Error: pila vacía');
            HALT;
    ...
END Desapilar;
```

Esta manera es demasiado drástica, pues no le da la posibilidad al usuario de remediar el error en situaciones perfectamente controlables; no tiene mucho sentido abortar un programa simplemente porque el operario haya introducido erróneamente el nombre de un fichero. En cambio, hay algunas situaciones concretas donde el error es irrecuperable y el programa debe acabar, como el error de protección de memoria que devuelve el sistema operativo cuando un puntero apunta a una zona de memoria prohibida para el programa.

Parámetros de control. También tenemos el tratamiento de errores a través de una variable que es introducida como parámetro de cada procedimiento, la cual toma un valor que puede ser analizado por el usuario y tomar las acciones que crea convenientes en cada circunstancia. El usuario ha de comprobar el valor de esta variable tras cada llamada a un procedimiento.

Por ejemplo:

```
TYPE    LISTAERROR = (SinError, ObjetoNoCreado, DivisionPorCero, SinMemoria);
PROCEDURE Asignar(VAR r:RACIONAL n,d:INTEGER;VAR error:LISTAERROR);
...
        error:=SinError;
        IF d = 0   THEN
                error:=DivisionPorCero
...
END Asignar;
```

La desventaja de esta opción es que se complica la interfaz de los procedimientos añadiendo un parámetro que no le es útil para sus cálculos, sino sólo para el control de errores. Esta política es muy usada en el lenguaje C, donde las funciones suelen tener un valor de vuelta que indica si ha habido error y cuál es, o en las órdenes de los sistemas operativos.

Variable de error encapsulada. Otro método de control de errores, que será el que empleemos a lo largo del presente curso, consiste en encapsular en la implementación del TAD una variable que nos indique las situaciones de error que se han presentado al

ejecutar las operaciones del TAD. Para ello, incluiremos en el módulo de definición un tipo enumerado TIPO_ERROR que abarque las distintas situaciones de error que pueden presentarse, así como una función Error de tipo TIPO_ERROR. Por ejemplo, en este TAD del número racional tendremos:

```
DEFINITION MODULE Racional;

    TYPE TIPO_ERROR=(SinError, ObjetoNoCreado, DivisionPorCero, SinMemoria);

    PROCEDURE Error (): TIPO_ERROR;
    .......
    .......
```

En el módulo de implementación del TAD definiremos una variable cod_error de tipo TIPO_ERROR, que se inicializará adecuadamente en el código del módulo de implementación. La función Error simplemente devolverá el valor de esta variable encapsulada:

```
IMPLEMENTATION MODULE Racional;

    cod_error : TIPO_ERROR;

    PROCEDURE Error (): TIPO_ERROR;
    BEGIN
        RETURN cod_error;
    END Error;
    ....
    .......
BEGIN

    cod_error:= SinError;

END Racional.
```

Cada uno de los procedimientos del TAD debe actualizar *en todas y cada una de sus ejecuciones* el valor de la variable cod_error con la finalidad de informar al usuario sobre la validez de los resultados obtenidos.

Finalmente, el tratamiento de errores se realizará en el programa de prueba a través de un procedimiento que podemos denominar VerError el cual llama a la función Error del TAD y, dependiendo del valor que devuelva esta función, tome las medidas oportunas. Después de cada llamada a cualquier procedimiento del TAD el usuario ha de comprobar qué ha ocurrido. Esto lo hará invocando al procedimiento VerError.

Para el tratamiento de errores en el modulo biblioteca (BibRac) será necesario importar del TAD tanto la función Error como el tipo TIPO_ERROR y asegurarse que no se ha producido error en las llamadas a procedimientos del TAD para poder continuar. Si se detecta que ha habido algún error, simplemente se acaba el procedimiento. El programa de prueba también tendrá que comprobar errores tras llamar a operaciones de la biblioteca.

TopSpeed Modula-2 ofrece un sistema parecido en algunos de sus módulos para el control de errores. En los módulos IO y FIO aparecen las variables globales OK y EOF. OK se actualiza tras cada operación e indica si se ha realizado correctamente o no. EOF se actualiza tras cada operación de lectura sobre un fichero e indica si se ha llegado al final del fichero o no. Este método tiene como desventaja que el usuario puede acceder directamente a las variables de error y manipularlas erróneamente poniendo, por ejemplo, la variable EOF a verdadero cuando realmente se está leyendo por la mitad del fichero.

Excepciones. En este método el segmento de código que ha detectado el error informa de él mediante la elevación de una excepción, análoga a las interrupciones Hardware o Software. Cuando se eleva una excepción se acaba la ejecución del procedimiento que la ha elevado y vuelve el control al procedimiento que la elevó. Este procedimiento puede tener definidas las acciones a tomar cuando ocurre esa situación excepcional mediante la definición de un manejador. En ese caso se dice que se ha capturado la excepción y se ejecuta el manejador. El bloque de código que ha capturado la excepción acaba. Si no ha definido el manejador, el procedimiento acaba y propaga la excepción al procedimiento que lo llamó. En la mayoría de los lenguajes, si llega alguna excepción al programa principal que éste no capture, se aborta la ejecución del programa.

Práctica Suplementaria.

Se trata de construir el tipo GRAN_NATURAL. Este tipo debe permitir almacenar naturales muy grandes, incapaces de ser almacenados en una variable CARDINAL normal del ordenador. Para ello el tipo GRAN_NATURAL será un tipo opaco definido en el módulo de implementación con la siguiente estructura:

```
TYPE
        CIFRAS = ARRAY [1 .. MAX_CIFRAS] OF DIGITO;
        NUMERO =    RECORD
                            validas: CARDINAL;
                            num: CIFRAS:
                    END;
        GRAN_NATURAL = POINTER TO NUMERO;
```

El módulo de definición debe incluir las siguientes operaciones:

```
DEFINITION MODULE GrNat;
CONST
        MAX_CIFRAS = << Un valor cualquiera >>;
TYPE
        GRAN_NATURAL;
        DIGITO = ['0' .. '9'];
        TIPO_ERROR = <<a definir >>;

PROCEDURE Error(): TIPO_ERROR;

PROCEDURE Crear():GRAN_NATURAL;

(* Pre: n ha sido creado y (1 ≤ Length(s) ≤ MAX_CIFRAS) y ('0'≤ s[i] ≤'9' ∀ i / 0 ≤ i < Length(s)) *)
        PROCEDURE Asignar(VAR n: GRAN_NATURAL;s: ARRAY OF CHAR);

        (* Pre: n ha sido creado. *)
        PROCEDURE Destruir(VAR n: GRAN_NATURAL);

        (* Pre: n ha sido creado y (1 ≤ i ≤  número de cifras de n) *)
        PROCEDURE Cifra(n: GRAN_NATURAL;i: CARDINAL):DIGITO;

        (* Pre: n ha sido creado *)
        PROCEDURE Longitud(n: GRAN_NATURAL): CARDINAL;
        END GrNat.
```

Haciendo uso de este TAD crear una biblioteca con las siguientes operaciones:

```
DEFINITION MODULE BibGNat;
FROM G_Nat IMPORT GRAN_NATURAL;

(* Pre: n1 y n2 han sido creados. *)
```

```
PROCEDURE Copiar(VAR n1: GRAN_NATURAL; n2: GRAN_NATURAL);

(* Pre: n, n1 y n2 han sido creados. *)
PROCEDURE Sumar(VAR n: GRAN_NATURAL;n1,n2: GRAN_NATURAL);

(* Pre: n, n1 y n2 han sido creados. *)
PROCEDURE Restar(VAR n: GRAN_NATURAL;n1,n2: GRAN_NATURAL);

(* Pre: n, n1 y n2 han sido creados. *)
PROCEDURE Multiplicar(VAR n: GRAN_NATURAL;n1,n2: GRAN_NATURAL);

(* Pre: n, n1 y n2 han sido creados y n2 <> 0 *)
PROCEDURE Dividir(VAR n: GRAN_NATURAL;n1,n2: GRAN_NATURAL);

(* Pre: n1 y n2 han sido creados. *)
PROCEDURE EsIgual(n1,n2: GRAN_NATURAL):BOOLEAN;
END BibGNat.
```

Por último, construir una pequeña calculadora sobre grandes naturales.