Práctica 8. TAD árbol binario de búsqueda

Objetivos.

Se trata de construir el TAD árbol binario de búsqueda y escribir una aplicación para obtener el listado de identificadores de un programa Modula-2.

Enunciado.

Construir el TAD ARBOL según el siguiente módulo de definición:

```
DEFINITION MODULE Arbol;

        FROM TadItem IMPORT ITEM;

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

        PROCEDURE Error(): TIPO_ERROR;

        PROCEDURE Crear(): ARBOL;
        PROCEDURE Destruir(VAR a: ARBOL);

        PROCEDURE EsVacio(a: ARBOL): BOOLEAN;
        PROCEDURE Esta(a: ARBOL; x: ITEM):BOOLEAN;
        (* pre: NOT EsVacio(a) *)
        PROCEDURE Raiz(a: ARBOL): ITEM;
        (* pre: NOT EsVacio(a) *)
        PROCEDURE Izquierdo(a: ARBOL): ARBOL;
        (* pre: NOT EsVacio(a) *)
        PROCEDURE Derecho(a: ARBOL): ARBOL;

        (* pre: NOT Esta(a, x) *)
        PROCEDURE Insertar(VAR a: ARBOL;x: ITEM);
        (* pre: Esta(a, x) *)
        PROCEDURE Borrar(VAR a: ARBOL;x: ITEM);

        PROCEDURE Copiar(VAR a1:ARBOL; a2:ARBOL);

END Arbol.
```

El TAD ARBOL se representará internamente mediante la siguiente estructura de datos:

```
TYPE    ARBOL = POINTER TO NODO;
        NODO  = RECORD
                        cont  :  ITEM;
                        izq,dch: ARBOL
                END;
```

Para implementar el TAD ARBOL de forma genérica, el módulo TadItem debe exportar una operación que nos permita comparar dos datos de tipo ITEM. Así, además del procedimiento ValorPorDefecto, incluiremos en el módulo de definición de TadItem el procedimiento

```
PROCEDURE Comparar(a, b: ITEM): INTEGER;
```

que devuelve -1 si a es menor que b, 0 si a es igual a b, y 1 si a es mayor que b.

Aplicación: Listado de identificadores para Modula-2.

Como aplicación, se trata de escribir un programa que lea un programa en Modula-2 y genere como resultado un listado en el que aparezcan en orden alfabético los identificadores definidos por el usuario. Un identificador es una cadena de caracteres que comienza por una letra y está compuesta por letras, dígitos y el carácter de subrayado. Se debe tener especial cuidado en no contabilizar como identificadores las cadenas contenidas dentro de un comentario o un literal de cadena; aunque en un primer prototipo este detalle se puede ignorar.

Lectura del código fuente

Para la lectura del código fuente, implementaremos una librería LexMod2 según el siguiente módulo de definición:

```
DEFINITION MODULE LexMod2;

    PROCEDURE Abrir(VAR nombre: ARRAY OF CHAR);
    PROCEDURE Cerrar;
    PROCEDURE SigIdent(VAR id: ARRAY OF CHAR);

END LexMod2.
```

En el módulo de implementación tendremos una variable encapsulada de tipo File, sobre la que se efectuarán las operaciones de lectura. Esta forma de abstracción se denomina **encapsulado de datos**: se oculta la definición de un dato (en nuestro caso, la variable de tipo File) y se interactúa con él a través de una interfaz. La variable de error del TAD es otro ejemplo de encapsulado de datos.

El procedimiento Abrir abre el fichero nombre que contiene el programa Modula-2 a procesar, mientras que el procedimiento Cerrar cierra este fichero.

El procedimiento SigIdent devuelve en id el siguiente identificador que aparece en el programa. Para ello, el procedimiento SigIdent va leyendo el fichero nombre carácter a carácter, agrupándolos hasta formar un *elemento lexicográfico* de Modula-2. Los elementos lexicográficos de Modula-2 pueden ser identificadores, constantes literales (enteros, reales, caracteres y cadenas), comentarios o símbolos. En general, puede saberse qué tipo de elemento lexicográfico estamos leyendo al encontrar su primer carácter: los números empiezan por un dígito, las cadenas por comillas simples o dobles, etc. Si el elemento lexicográfico *no* es un identificador, se descarta y se continúa leyendo el fichero. Por el contrario, si el elemento lexicográfico es un identificador, se devuelve en el parámetro id. Por ejemplo, las primeras llamadas aplicadas al fichero LEXMOD2.DEF devolverán DEFINITION, MODULE, LexMod2, PROCEDURE, Abrir, VAR, nombre, ARRAY.... El fin de fichero se señalará devolviendo en id la cadena vacía.

Generación del listado de identificadores

Para generar el listado de identificadores, se utilizará el TAD ARBOL empleando el siguiente tipo ITEM:

```
CONST MAX_IDENTIFICADOR= <<a definir>>;
TYPE ITEM= ARRAY [1..MAX_IDENTIFICADOR] OF CHAR;
```

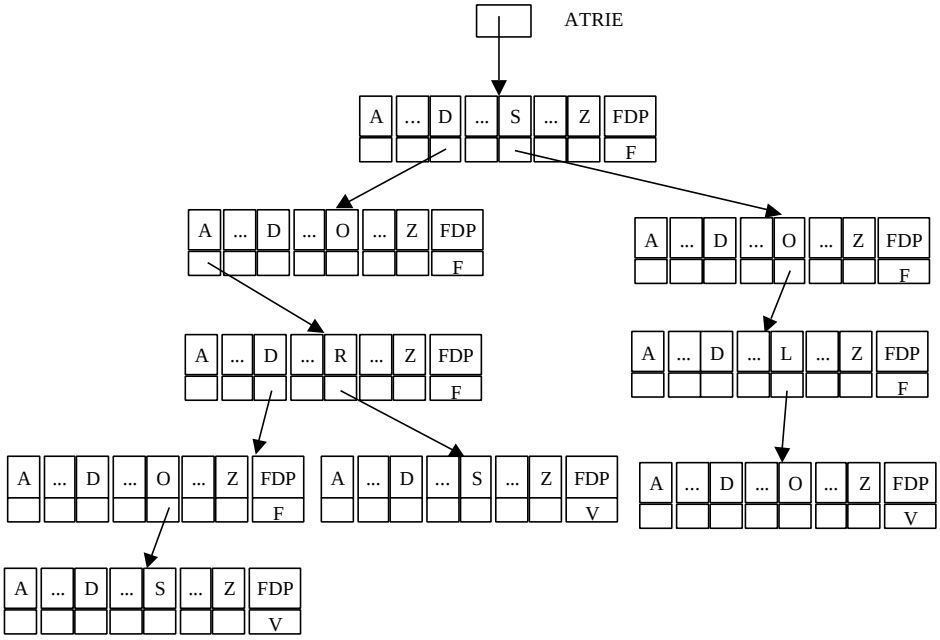
Antes de incluir un identificador en el ARBOL se debe comprobar que no sea una palabra reservada de Modula-2 (en el listado sólo deben aparecer los identificadores definidos por el usuario). Si se trata de la primera aparición del identificador, se inserta en el ARBOL.

Impresión del listado de identificadores

Para imprimir el listado, basta efectuar un recorrido *inorden* del ARBOL.

Práctica Suplementaria

Un árbol TRIE es un árbol general, es decir, cada nodo puede tener un número diferente de hijos, que es muy apropiado para representar diccionarios de palabras. Las palabras vienen representadas por caminos dentro del árbol. Para evitar tomar como palabra independiente las primeras letras de otra palabra, se añade un campo especial de final de palabra (FDP). Así podemos distinguir si, por ejemplo, DAD es parte de DADO o es una palabra por sí solo.



Este TRIE contiene las palabras: DADO, DAR y SOL.

```
(* Devuelve un árbol de tipo ATRIE vacía *)
PROCEDURE Crear (): ATRIE;

(* Inserta una palabra en el árbol trie. Si ya está no se hace nada. *)
PROCEDURE Insertar(VAR a: ATRIE; s: ARRAY OF CHAR);

(* Elimina una palabra en el árbol trie. Si no está no se hace nada. *)
PROCEDURE Eliminar(VAR a: ATRIE; s: ARRAY OF CHAR);

(* Devuelve TRUE si la palabra está en el árbol, FALSE en caso contrario *)
PROCEDURE Está( a: ATRIE; s: ARRAY OF CHAR): BOOLEAN;

(* Devuelve TRUE si el árbol no tiene elementos. FALSE en caso contrario *)
PROCEDURE EsVacío(a: ATRIE): BOOOLEAN;

(* Devuelve la palabra anterior en el árbol. Si la palabra que se pasa de parámetro es la
   primera, se devuelve ella misma. *)
PROCEDURE Anterior(a: ATRIE; p: ARRAY OF CHAR; VAR ant: ARRAY OF CHAR);
```

```
(* Devuelve la palabra siguiente en el árbol. Si la palabra que se pasa de parámetro es la
   última, se devuelve ella misma.*)
PROCEDURE Siguiente(a: ATRIE; p: ARRAY OF CHAR; VAR sig: ARRAY OF CHAR);

(* Libera toda la memoria ocupada por el árbol *)
PROCEDURE Destruir(VAR a: ATRIE);

(* Lista por pantalla todos las palabras contenidas en el árbol *)
PROCEDURE Listar(a: ATRIE);
```