PROYECTO BASE DE DATOS COMANDO

CONTENIDO

INTRODUCCIÓN

"La disciplina es el puente entre las metas y los logros."

Jim Rohn

El siguiente proyecto es extremadamente simple en comparación a los anteriores, ya que está pensado para que entrenes lo visto con **archivos**, pudiendo crear, modificar y eliminar **ficheros** tal como has visto en las clases.

Implementarás un sistema de base de datos con algunos cambios respecto a lo que vimos anteriormente, encargándote de toda la gestión de archivos. Es un entrenamiento sencillo pero efectivo, que deberás llevar a cabo para avanzar en el curso, ya que de ahora en más, la persistencia de datos será algo habitual en las lecciones siguientes.

OBJETIVO

Crear un programa en Pascal que implemente el sistema que se describirá a continuación, practicando el tratamiento de **archivos**. El estudiante deberá poder implementar el programa en cuestión haciendo uso de todas las herramientas vistas hasta ahora en el curso, pudiendo consultar a los docentes a través de la plataforma (***Udemy*** o ***VirtuaEdu***).

Este proyecto tiene como objetivo afianzar todos los conceptos que el estudiante ya aprendió en las clases, enfatizando el uso de ***archivos con tipo***.

BASE DE DATOS COMANDO

El programa **Base de Datos Comando** permitirá gestionar una pequeña base de datos haciendo uso de un sistema de comandos. Lo único que se podrá guardar en el sistema son registros de personas con los siguientes datos:

- ✓ **Id**: Un número único que indicará el índice del registro en el archivo de base de datos.
- ✓ **Documento**: Una cadena de caracteres única para la persona a registrar.
- ✓ **Nombre**: El nombre de la persona. Puede ser compuesto, por ejemplo: "María del Carmen".
- ✓ **Apellido**: El apellido de la persona. Puede ser compuesto, por ejemplo: "De Leon".
- ✓ **Edad**: La edad de la persona en formato numérico.
- ✓ **Peso**: El peso de la persona en formato numérico.

El sistema permitirá ***crear***, ***modificar***, ***eliminar*** y ***buscar*** registros, así como ***optimizar*** el uso de espacio en el disco duro y ***mostrar*** información sobre la cantidad de registros activos. Todo esto se detallará a continuación.

SISTEMA DE COMANDOS

Hasta ahora en los ejemplos vistos hemos leído desde la entrada estándar un dato a la vez, línea por línea. Ahora esto cambiará, utilizando un sistema de comandos que permita ingresar todo en una sola línea de entrada.

Los comandos son ampliamente utilizados hoy día en la mayoría de los sistemas. El usuario estándar los desconoce y piensa que ya no son usados, acostumbrado a las ventanas gráficas, sin embargo, todo sistema operativo los tiene y utiliza, así como la gestión de servidores de Internet, programas de diseño, programas de gestión de bases de datos, entre muchísimos otros.

Muchos sistemas Linux, por ejemplo, no incluyen interfaz gráfica, y es común que los servidores tampoco lo hagan, quedándose solamente con una consola de comandos. ¿Por qué? Porque la interfaz gráfica requiere muchos recursos para funcionar (espacio en disco, memoria RAM, tiempo de ejecución para funcionar y dibujarse, etc.) Por tanto, al quitar la interfaz gráfica todos los recursos físicos se enfocan en lo importante: ejecutar los programas para los usuarios.

Un sistema de comandos normalmente tiene el siguiente formato:

```
NOMBRE_DEL_COMANDO [PARAMETRO_1] [PARAMETRO_2] … [PARAMETRO_N]
```

Un **COMANDO** en este modelo no es otra cosa que una línea de texto. Por tanto, un comando básicamente separa las palabras de la entrada estándar. La primera palabra es el nombre del comando, y el resto sus parámetros. Por ejemplo, en *Windows*, el comando para copiar directorios en la consola se llama **XCOPY**, y su formato es:

```
XCOPY DIRECTORIO_ORIGINAL DIRECTORIO_DESTINO
```

Recibe por tanto dos parámetros, ***el directorio de origen*** (desde donde vamos a copiar), y el directorio de destino (hacia dónde vamos a copiar y qué). También puede recibir más parámetros aún, que indicarán permisos de lectura y escritura entre otros detalles. En *Linux*, el comando de copiado en vez de llamarse **XCOPY** se llama **CP**, pero usa el mismo formato:

```
cp DIRECTORIO_ORIGINAL DIRECTORIO_DESTINO
```

Un último ejemplo puede ser un comando del gestor de base de datos *MySQL* usado para crear una nueva tabla de datos, el cual se llama **CREATE TABLE**:

```
CREATE TABLE NOMBRE_TABLA ([LISTA_DE_COLUMNAS])
```

Muchos parámetros pueden requerir texto con espacios, y eso es un problema, porque el espacio se usa como separador, y por tanto cada espacio que se escribe en la línea de comandos indica que hay un parámetro nuevo. Para solucionar esto se usan las comillas, encerrando un parámetro en ellas para indicar que se quiere tener todo lo que hay dentro como un único parámetro. Por ejemplo, veamos el comando que sirve para cambiar de directorio que, tanto en *Windows* como en *Linux*, se llama **CP** (*Change Directory*):

```
cd C:\Juegos\Buscaminas
```

Ahora bien ¿Qué pasa cuando una carpeta contiene espacios en su nombre? Por ejemplo, si en vez de "`Juegos`" se llama "`Mis juegos`":

```
cd C:\Mis juegos\Buscaminas
```

El comando anterior no funcionará, porque el espacio en blanco funciona como separador, y por tanto habría dos parámetros: "**`C:\Mis`**" y "**`juegos\Buscaminas`**". Para solucionar esto, tanto en *Windows* como en *Linux*, ha de usarse las comillas en el texto para indicar que TODO es un solo parámetro:

```
cd "C:\Mis juegos\Buscaminas"
```

De este modo, al encerrar entre comillas, el comando cd entiende que se trata de un parámetro compuesto y toma todo el texto que está dentro de las comillas.

El sistema que implementaremos hará uso de una línea de comandos para agilizar el ingreso de datos.

EL PROGRAMA

El sistema que programarás en este proyecto no hará uso de una interfaz gráfica de usuario, sino que como ya se explicó, utilizará la línea de comandos implementando un ***sistema de comandos*** para agilizar el ingreso de datos.

ESPECIFICACIÓN

Al abrir el programa se verá simplemente un **prompt** indicando que se requiere algún dato de entrada:

```
>>
```

Luego del símbolo **>>** habrá un espacio en blanco.
El usuario deberá ingresar un comando válido. Si no se ingresa uno correcto se mostrará el mensaje:

```
ERROR: El comando ingresado no es correcto
```

Luego se volverá a mostrar el **`prompt`**.
No se limpiará la pantalla, sino que se dejará siempre visible lo que se ha escrito:

```
>>
ERROR: El comando ingresado no es correcto
>>
```

A continuación listaremos los comandos y sus parámetros.

COMANDOS

Como ya se mencionó el sistema hará uso de comandos específicos que definiremos en detalle.

COMANDO **NUEVO**

Para ingresar un nuevo registro a la base de datos se utilizará el comando **`NUEVO`**, el cual recibirá como parámetros los datos de la persona a guardar. El formato es el siguiente:

```
NUEVO DOCUMENTO NOMBRE APELLIDO EDAD PESO
```

Por ejemplo:

```
NUEVO 12345 Vladimir Rodriguez 30 72
```

El comando anterior creará un registro con el **documento** `12345`, **nombre** `Vladimir`, **apellido** `Rodriguez`, **edad** `30` y **peso** `72`.

El comando NUEVO admitirá nombres y apellidos compuestos, como “Maria del Carmen” o “De Leon”. Para ello ha de utilizarse comillas dobles:

```
NUEVO 12345 “Maria del Carmen” “De Leon” 56 68
```

El comando anterior creará un registro con el **documento** `12345`, **nombre** `Maria del Carmen`, **apellido** `De Leon`, **edad** `56` y **peso** `68`.

CONTROL DE ERRORES

El comando controlará la cantidad de argumentos (**5**), si esto es incorrecto se mostrará el mensaje:

```
ERROR: Cantidad de parametros incorrecta: [DOCUMENTO,NOMBRE,APELLIDO,EDAD,PESO].
```

Si la cantidad de argumentos es correcta se verificará que no exista ya registrada otra persona con el mismo documento que se ha ingresado. En caso de que esto ocurra se mostrará el mensaje:

```
ERROR: Ya existe un registro con documento DOCUMENTO: NOMBRE APELLIDO
```

Veamos un ejemplo:

```
NUEVO 12345 Vladimir Rodriguez 30 72
Registro agregado exitosamente
NUEVO 12345 "Maria del Carmen" Cabrera 30 72
ERROR: Ya existe un registro con documento 12345: Vladimir Rodriguez
```

Como se puede ver, en el ejemplo anterior se intentó registrar a dos personas: primero a **Vladimir Rodriguez** con documento **12345**, el cual se agregó correctamente (se puede ver el mensaje resaltado en verde), luego se intentó agregar a **María del Carmen** con el mismo documento, y como este dato ha de ser único, el sistema no permitió hacerlo, mostrando el error y además indicando a quién pertenece dicho documento.

Si no existe una persona ya registrada con el documento ingresado, y la cantidad de parámetros es correcta, se verificará que **EDAD** sea un valor numérico entero, en caso contrario se mostrará:

```
ERROR: El parámetro EDAD debe ser un numero entero.
```

Si esto no ocurre, se verificará lo mismo para el parámetro **PESO**. En caso de que error se mostrará:

```
ERROR: El parámetro PESO debe ser un numero entero.
```

Finalmente, si todo está bien, se mostrará el mensaje:

```
Registro agregado exitosamente
```

COMANDO **MODIFICAR**

Este comando permite, como su nombre lo indica, **MODIFICAR** un registro existente en la base de datos. Recibe 5 argumentos, tal como se muestra a continuación:

```
MODIFICAR DOCUMENTO N_NOMBRE N_APELLIDO N_EDAD N_PESO
```

Recibe por tanto el **DOCUMENTO** original de la persona que se quiere modificar, y luego los nuevos datos. El formato es exactamente el mismo que para el comando **NUEVO**.

CONTROL DE ERRORES

Primero se verifica que la cantidad de parámetros sea **5**, en caso negativo se muestra:

```
ERROR: Cantidad de parametros incorrecta: [DOC_ORIGINAL,NOMBRE,APELLIDO,EDAD,PESO].
```

Luego verifica que **exista** y **esté activo** un registro con el documento indicado. En caso negativo muestra:

```
No existe un registro con documento DOCUMENTO para modificar.
```

Por ejemplo:

```
MODIFICAR 3325 Juan Jose 25 74
No existe un registro con documento 3325 para modificar.
```

Al igual que para el comando NUEVO se verificará que EDAD y PESO sean números enteros, en el mismo orden, y en caso negativo se mostrarán los mismos mensajes de error.

Finalmente, si no hay ningún problema, se mostrará el mensaje:

```
Modificacion exitosa
```

COMANDO **ELIMINAR**

Permite eliminar un registro en la base de datos sin borrarlo del archivo. Este comando lo que hace es **marcar al registro como eliminado**, pero dicho registro permanecerá allí. Sin embargo, cuando un registro ha sido marcado como eliminado no aparecerá en búsquedas, no se podrá modificar porque se tomará como inexistente, y se podrá crear un nuevo registro con el mismo número de documento.

Este comando, a diferencia de los anteriores, puede recibir 1 o 2 parámetros dependiendo de si se quiere eliminar un registro específico o TODA la base de datos. En la primera forma elimina un registro específico, y recibe como primer parámetro la cadena **-D**, y luego el **documento** a borrar:

```
ELIMINAR -D 12345
```

En la segunda forma recibe solo un parámetro el cual debe ser -T:

```
ELIMINAR -T
```

Tanto **-D** como **-T** son valores predefinidos y deben usarse correctamente.

CONTROL DE ERRORES

Al ingresar este comando lo primero que se verifica es que la cantidad de parámetros sea **1** o **2**, sino:

```
ERROR: Cantidad de parametros incorrecta: [-T] o [-D,DOCUMENTO]
```

Para ambos formatos de este comando, luego se verifica que el primer parámetro sea válido, es decir, que sea **-D** o **-T** según corresponda. Si esto no es así se muestra:

```
ERROR: El argumento no es correcto o faltan datos
```

En caso de que el primer argumento sea -D se esperará luego el DOCUMENTO del registro a eliminar, por tanto se verificará que exista en la base de datos una persona con el documento indicado (y que no haya sido marcada como eliminada anteriormente). Si no encuentra un registro activo con el documento pasado como parámetro, se mostrará el mensaje:

```
ERROR: No hay un registro con documento DOCUMENTO para eliminar.
```

En caso de que se ingrese el parámetro -T para borrar toda la base de datos no hay nada que verificar. Se marcan todos los registros como eliminados y no se muestra ningún mensaje de confirmación.

```
>> ELIMINAR -T
>>
```

En caso de eliminar un documento específico, si no hubo errores se muestra:

```
Registro eliminado exitosamente
```

COMANDO **BUSCAR**

Tiene dos formas, una en la cual mostrará todos los registros existentes y activos (no eliminados) en la base de datos, y otra en la que mostrará un registro específico. La primera forma se escribe:

```
BUSCAR
```

No recibe argumentos. Su salida estará formateada para mostrar una tabla. Si la base de datos está vacía se mostrará lo siguiente:

```
    ID|  DOCUMENTO|               NOMBRE|             APELLIDO|  EDAD|  PESO
-------------------------------------------------------------------------------
Registros mostrados: 0
```

Si la base de datos contiene uno o más registros se mostrarán en el mismo formato:

```
    ID|  DOCUMENTO|               NOMBRE|             APELLIDO|  EDAD|  PESO
-------------------------------------------------------------------------------
     2|      12345|             Vladimir|            Rodriguez|    30|    72
     3|      12346|               Lorena|               Cuello|    30|    56
     4|       2356|     Maria del Carmen|        De los Santos|    55|    66
Registros mostrados: 3
```

Al final siempre se mostrará la cantidad de registros existentes.

La salida con formato de tabla ya estará programada para el estudiante, ya que el foco de este proyecto está en poder implementar los comandos aquí descritos y manipular el archivo de base de datos. No es la idea perder tiempo en otras cuestiones inherentes a *programación estructurada* porque esto ya ha sido debidamente entrenado en todos los módulos anteriores.

La otra forma del comando BUSCAR recibe un solo argumento, indicando el DOCUMENTO deseado:

```
BUSCAR DOCUMENTO
```

En este caso se muestra únicamente el registro con el documento indicado. Por ejemplo:

```
>> BUSCAR 12345
    ID|  DOCUMENTO|               NOMBRE|             APELLIDO|  EDAD|  PESO
-------------------------------------------------------------------------------
     2|      12345|             Vladimir|            Rodriguez|    30|    72
>>
```

CONTROL DE ERRORES

Si se ingresa más de 1 parámetro para este comando se mostrará el mensaje:

```
La cantidad de parametros es incorrecta: [] o [DOCUMENTO]
```

Si se ingresa un documento que no existe o fue eliminado, se mostrará:

```
No existe un registro con DOCUMENTO [DOCUMENTO]
```

COMANDO ESTADOSIS

ESTADOSIS (estado sistema) no recibe ningún parámetro y muestra simplemente tres datos en la salida estándar:

```
Registros totales: [FILESIZE] - Registros activos: [NO ELIMINADOS] - Registros eliminados: [ELIMINADOS]
```

Básicamente indica cuántos registros hay en el archivo de base de datos, contando todo lo que allí exista, tanto los eliminados como los activos. Luego muestra la cantidad de registros activos, es decir, que no fueron eliminados, y los que sí fueron eliminados. Por ejemplo:

```
Registros totales: 5 - Registros activos: 3 - Registros eliminados: 2
```

En este caso hay 5 registros en el archivo, 3 de ellos activos y 2 eliminados.

CONTROL DE ERRORES

Simplemente se verifica que no se ingrese ningún parámetro para este comando. Si se ingresa uno o más parámetros se mostrará el mensaje:

```
La cantidad de parametros es incorrecta []
```

COMANDO **OPTIMIZAR**

Este comando lleva a cabo una eliminación real de los registros en el archivo de base de datos, limpiándolo de forma definitiva. La tarea que se lleva a cabo es la que ya hemos visto en clases anteriores:

1. Se crea un archivo temporal nuevo.
2. Se copian al archivo nuevo los registros del archivo original que no estén marcados como eliminados.
3. Se borra el archivo original.
4. Se renombra el archivo temporal para que sea ahora la base de datos del sistema.

Este comando no recibe parámetros y tampoco los controla, es decir, ignora cualquier parámetro que se ingrese. Las siguientes líneas son equivalentes:

```
OPTIMIZAR
```

```
OPTIMIZAR REGISTROS
```

```
OPTIMIZAR MI ABUELA
```

Se podría hacer que **ESTADOSIS** funcionara igual, pero lo hemos hecho así a propósito para que el estudiante pueda ver las diferencias de hacer una cosa u otra.

Se mostrará el mensaje:

```
Base de datos optimizada
```

COMANDO **SALIR**

Simplemente cierra la aplicación sin mostrar ningún mensaje. No recibe parámetros y si el usuario aun así los ingresara serán ignorados, tal como **OPTIMIZAR**.

```
SALIR
```

INGRESO DE COMANDO INEXISTENTE

Si el usuario ingresa un comando que no existe se mostrará el mensaje:

```
ERROR: El comando ingresado no es correcto
```

Se podría hacer que se mostraran los comandos existentes y sus formatos, como hacen la mayoría de los sistemas, pero eso lo dejamos a elección del estudiante ya que no aporta al objetivo de este proyecto.

UNIDAD **UCOMANDO**

Como se verá más adelante en este documento, el programa se dividirá en dos archivos, la unidad `UComando` y el programa principal `BaseDeDatosSimple`. La unidad `UComando` ya estará completamente programada para facilitar la vida del estudiante. Su función básicamente es recibir una cadena de caracteres (`String`) desde la entrada estándar y obtener de ella el nombre del comando y los parámetros. Por ejemplo, si recibe la cadena: "**CREACION Celula Tejido Persona 23 5689 78**" se desglosaría así:

- Nombre del comando: **`CREACION`**
- Parámetro 1: **`Celula`**
- Parámetro 2: **`Tejido`**
- Parámetro 3: **`Persona`**
- Parámetro 4: **`23`**
- Parámetro 5: **`5689`**
- Parámetro 6: **`78`**

La unidad `UComando` funcionará con cualquier cadena de caracteres, sin importar si se refiere a uno de los comandos descritos en este documento o no, ya que su propósito es general, y por tanto sirve para programar tanto este proyecto como cualquier otro programa que funcione con la misma mecánica.

A continuación se describe la forma en que funciona esta unidad para que el estudiante sepa cómo utilizarla para la realización de este proyecto.

CONSTANTES

Esta unidad define dos constantes que para el estudiante no son necesarias, ya que hará uso de las operaciones disponibles en la interfaz de la unidad. Sin embargo las describimos brevemente aquí:

```
const
  MAX_PARAM_COMANDO= 25; //Máximo número de parámetros admitidos.
  COMILLAS= #34; //El carácter de comillas.
```

No requieren mucho más tratamiento aquí.

TIPOS DE DATOS

Esta unidad define tes tipos de datos y solo dos son importantes para el estudiante. El primero es:

```
TParametro= record
      datoString: String;
      datoNumerico: integer;
      esNumero: boolean;
  end;
```

Simplemente transformamos un parámetro cualquiera en un dato más completo, indicando si es o no un número (`esNumero`) y almacenando su información (`datoString` y `datoNumerico`). Siguiendo el ejemplo anterior, para la cadena **CREACION Celula Tejido Persona 23 5689 78** tendríamos que:

- **Para el parámetro `Celula`:**
  - `datoString= 'Celula'`
  - `datoNumerico= 0` (o cualquier otro, no importa)
  - `esNumero= false`

- **Para el parámetro `23`:**
  - `datoString= '23'`
  - `datoNumerico= 23`
  - `esNumero= true`

De esta forma será muy simple comprobar si un parámetro es numérico o no, y obtener sus valores, tanto en formato `String` como en formato `Integer`.

```
TComando= record
      nombreComando: String;
      haySiguiente: boolean;
      listaParametros: TListParametros;
      puntero: byte;
  end;
```

Este tipo define a comando en sí mismo, almacenando su nombre, y su lista de parámetros. Para el caso visto anteriormente **`nombreComando`** sería **CREACION**, y luego tendríamos los 6 parámetros en el orden en que son ingresados.

Los parámetros se podrán leer de forma secuencial, tal como sucede con un archivo tipado. Primero se obtendrá el primer parámetro, luego el segundo, y así hasta el final.

Mediante este modelo, es posible entonces tener fácilmente el nombre del comando ingresado en formato `String` y luego la lista de parámetros usando el tipo `TParametro` descrito anteriormente.

NOTA IMPORTANTE: El estudiante no debe acceder a los atributos internos de estos registros, sino que simplemente deberá hacer uso de las operaciones que se describirán a continuación, ya que éstas proveen de todo lo necesario para implementar el proyecto e incluso para implementar cualquier programa que trabaje con un modelo de comandos.

OPERACIONES PARA TPARAMETRO

Como se indica anteriormente, el estudiante solo deberá utilizar las operaciones provistas en esta unidad ya que están diseñadas específicamente para brindar funcionalidad a los tipos **TParametro** y **TComando**, de forma tal que con ellas el estudiante pueda trabajar con estas estructuras de forma simple.

A continuación se describen las 4 operaciones provistas para trabajar con **TParametro**.

FUNCIÓN ESPARAMETROSTRING

```
{Retorna TRUE si el parámetro p es de tipo numérico, FALSE si no.
 Esta operación es análoga a esParametroNumerico}
function esParametroString(const p: TParametro): boolean;
```

Como indican los comentarios, recibe un argumento de tipo `TParametro` e indica si éste es un parámetro de tipo `String` o no. Básicamente indica si la cadena de caracteres contiene solo números o no. De esta forma se puede distinguir fácilmente si tenemos un entero numérico.

Por ejemplo, para el parámetro **Celula** retornaría `TRUE`, y para el parámetro **23** retornaría `FALSE`.

FUNCIÓN ESPARAMETRONUMERICO

```
{Retorna TRUE si el parámetro p es de tipo numérico, FALSE si no.
Esta operación es análoga a esParametroString}
function esParametroNumerico(const p: TParametro): boolean;
```

Análogamente a la operación anterior, esta función indica si el parámetro que se pasa como argumento es numérico, es decir, si solo contiene números enteros. No tiene mucho más misterio.

Por ejemplo, para el parámetro **Celula** retornaría `FALSE` y para el parámetro **23** retornaría `TRUE`, justamente lo inverso a `esParametroString`.

FUNCIÓN OBTENERNUMERO

```
{En caso de que p sea un parámetro de tipo numérico, asigna a n el
número contenido en p y retorna TRUE. De lo contrario, retorna FALSE.}
function obtenerNumero(var n: byte; const p: TParametro): boolean;
```

Esta operación retornará el número contenido en el parámetro, solamente si es numérico, y lo asignará al argumento **n** que se pasa por referencia. Siguiendo el ejemplo anterior, para el parámetro **Celula** retornaría `FALSE` y por tanto el argumento **n** quedaría con un valor indefinido; para el parámetro **23** retornaría `TRUE` y **n** justamente quedaría valiendo `23`.

FUNCIÓN OBTENERSTRING

```
{Retorna el contenido de un parámetro como un String, sin importar
si el parámetro p es numérico o no, siempre se obtiene un String.
Si, por ejemplo, el parámetro es numérico y tiene el valor 89, se
obtenddrá el String '89'.}
function obtenerString(const p: TParametro): String;
```

Esta función retorna el contenido del parámetro en formato `String`. Por ejemplo, para el parámetro Celula retornaría `'Celula'` y para el parámetro 23 retornaría `'23'`. Como se puede ver, no importa si el parámetro es numérico o no, siempre retornará su contenido como un `String`.

OPERACIONES PARA TCOMANDO

El tipo **TComando** es justamente el que permitirá trabajar con el sistema de comandos descrito en este documento, y para su tratamiento se han definido 6 operaciones. Todas ellas permitirán implementar el comportamiento descrito en este documento e incluso diseñar otros programas que usen comandos como forma de ingreso de datos.

FUNCIÓN CREARCOMANDO

```
function crearComando(entrada: String): TComando;
```

Esta función recibe como argumento una cadena de caracteres (`String`) y la procesa para distinguir el nombre del comando y sus parámetros, retornando así un registro **`TComando`** ya listo para trabajar. Por ejemplo, si recibe como argumento la cadena: **'CREACION Celula Tejido Persona 23 5689 78'**, el nombre del comando será **CREACION** y luego tendremos la lista de 6 parámetros bien definidos, todos de tipo **`TParametro`**, por lo tanto, podremos aplicar las operaciones descritas anteriormente.

FUNCIÓN NOMBRECOMANDO

```
{Retorna el nombre del comando c como un String}
function nombreComando(const c: TComando): String;
```

Recibe algo de tipo `TComando` y retorna su nombre en formato `String`. Una vez que `crearComando` retornó algo de tipo `TComando`, esta operación retornará el nombre del mismo en formato `String`.

FUNCIÓN HAYSIGUIENTEPARAMETRO

```
{Retorna TRUE si hay al menos un parámetro más por ser leído en la
lista de parámetros de c, FALSE si no hay ninguno.}
function haySiguienteParametro(const c: TComando): boolean;
```

Como se explicó anteriormente, los parámetros de un comando se guardan en una lista de forma secuencial, por tanto, solo se pueden recorrer en orden, del primero al último, uno por vez. Esta lista puede incluso estar vacía (no tener parámetros). Siguiendo el ejemplo anterior, tendríamos que para la cadena **'CREACION Celula Tejido Persona 23 5689 78'** sus parámetros quedarían así:

Parametro1	Parametro2	Parametro3	Parametro4	Parametro5	Parametro6
datoString 'Celula'	datoString 'Tejido'	datoString 'Persona'	datoString '23'	datoString '5689'	datoString '78'
datoNumerico 0	datoNumerico 0	datoNumerico 0	datoNumerico 23	datoNumerico 5689	datoNumerico 78
esNumero **FALSE**	esNumero **FALSE**	esNumero **FALSE**	esNumero **TRUE**	esNumero **TRUE**	esNumero **TRUE**

La función **`haySiguienteParametro`** indicará si hay un parámetro más por ver según al que se esté apuntando, retornando **`TRUE`** en caso, o **`FALSE`** en caso contrario. Cuando un comando es creado, el puntero de parámetros está posicionado justo antes del primer parámetro. En la siguiente imagen, la barra roja indica la posición de puntero. Como delante de él está **`Parametro1`**, entonces `haySiguienteParametro` retorna **`TRUE`**:

Parametro1	Parametro2	Parametro3	Parametro4	Parametro5	Parametro6
datoString 'Celula'	datoString 'Tejido'	datoString 'Persona'	datoString '23'	datoString '5689'	datoString '78'
datoNumerico 0	datoNumerico 0	datoNumerico 0	datoNumerico 23	datoNumerico 5689	datoNumerico 78
esNumero **FALSE**	esNumero **FALSE**	esNumero **FALSE**	esNumero **TRUE**	esNumero **TRUE**	esNumero **TRUE**

Se retornará FALSE cuando el puntero de parámetros esté, en este caso, sobre **`Parametro6`** porque no hay nadie delante de él, o bien cuando la lista de parámetros es vacía.

FUNCIÓN SIGUIENTEPARAMETRO

```
function siguienteParametro(var c: TComando): TParametro;
```

Esta operación devuelve el siguiente parámetro disponible en la lista de parámetros. Es necesario que `haySiguienteParametro` retorne TRUE para invocarla.

Siguiendo el ejemplo anterior, tenemos la lista de 6 parámetros del comando **CREACION**, e inicialmente el puntero de parámetros está posicionado justo antes del inicio de la lista:

Parametro1	Parametro2	Parametro3	Parametro4	Parametro5	Parametro6
datoString 'Celula'	datoString 'Tejido'	datoString 'Persona'	datoString '23'	datoString '5689'	datoString '78'
datoNumerico 0	datoNumerico 0	datoNumerico 0	datoNumerico 23	datoNumerico 5689	datoNumerico 78
esNumero FALSE	esNumero FALSE	esNumero FALSE	esNumero TRUE	esNumero TRUE	esNumero TRUE

Al invocar a **siguienteParametro**, el puntero se mueve una posición adelante, quedando ahora posicionado en **Puntero1** y además se retorna **Puntero1** como resultado:

Parametro1	Parametro2	Parametro3	Parametro4	Parametro5	Parametro6
datoString 'Celula'	datoString 'Tejido'	datoString 'Persona'	datoString '23'	datoString '5689'	datoString '78'
datoNumerico 0	datoNumerico 0	datoNumerico 0	datoNumerico 23	datoNumerico 5689	datoNumerico 78
esNumero FALSE	esNumero FALSE	esNumero FALSE	esNumero TRUE	esNumero TRUE	esNumero TRUE

Si volvemos a invocar a **siguienteParametro**, ahora el puntero se moverá un lugar hacia adelante ubicándose en **Parametro2** y además lo retorna como resultado:

Parametro1	Parametro2	Parametro3	Parametro4	Parametro5	Parametro6
datoString 'Celula'	datoString 'Tejido'	datoString 'Persona'	datoString '23'	datoString '5689'	datoString '78'
datoNumerico 0	datoNumerico 0	datoNumerico 0	datoNumerico 23	datoNumerico 5689	datoNumerico 78
esNumero FALSE	esNumero FALSE	esNumero FALSE	esNumero TRUE	esNumero TRUE	esNumero TRUE

De este modo se puede ir recorriendo la lista de parámetros en el orden en que éstos se ingresaron en la entrada estándar, ya que lo que se pasará a la operación **crearComando** será justamente la cadena de caracteres que el usuario ingresó.

PROCEDIMIENTO RESETEARPARAMETROS

```
procedure resetearParametros(var c: TComando);
```

Reestablece el puntero de parámetros justo una posición antes del inicio de la lista, tal como cuando está al ser creada. Su función es que se pueda volver a recorrer la lista de parámetros desde el principio si fuera necesario, ya que solo se puede avanzar hacia adelante.

FUNCIÓN CANTIDADPARAMETROS

```
function cantidadParametros(const c: TComando): byte;
```

Tal como su nombre lo indica, retorna la cantidad de parámetros que tiene el comando **c**. Para nuestro ejemplo, se retornaría el valor 6 porque hay 6 parámetros.

IMPLEMENTACIÓN

Al igual que en proyectos anteriores, este programa se dividirá en más de un archivo, en concreto dos:

- **Unidad** `UComando.pas`: Contiene los tipos de datos y operaciones para poder trabajar con comandos tal como ha sido descrito en este documento. Estará completamente programada para el estudiante, no es parte del proyecto implementar sus operaciones, sino usarlas.

- **Programa principal** `BaseDeDatosSimple.lpr`: Contiene al programa principal y todo lo que el estudiante debe programar. Habrá algunas rutinas (funciones y procedimientos) ya programados para facilitar la implementación al estudiante.

El estudiante deberá utilizar la unidad **`TComando`** y las estructuras que se describirán a continuación para llevar adelante su proyecto.

DEFINICIONES

Veamos entonces las definiciones de constantes, tipos y operaciones que utilizarás en este proyecto, todas dentro del programa principal, que es donde trabajarás en esta oportunidad.

CONSTANTES

```
(*Lista de comandos en String*)
COMANDO_NUEVO_TEXTO= 'NUEVO';
COMANDO_MODIFICAR_TEXTO= 'MODIFICAR';
COMANDO_ELIMINAR_TEXTO= 'ELIMINAR';
COMANDO_BUSCAR_TEXTO= 'BUSCAR';
COMANDO_OPTIMIZAR_TEXTO= 'OPTIMIZAR';
COMANDO_ESTADOSIS_TEXTO= 'ESTADOSIS';
COMANDO_SALIR_TEXTO= 'SALIR';
PARAMETRO_ELIMINAR_DOC= '-D';
PARAMETRO_ELIMINAR_TODO= '-T';
```

Definen simplemente los `String` que identifican a los comandos de este sistema en particular, de este modo se usarán luego para comparar la entrada del usuario con estos valores y determinar qué comando específicamente fue ingresado. El estudiante no hará uso de ellas ya que esta parte del sistema estará programada y funcional.

```
  (*Formatos para imprimir datos de salida como una tabla*)
  FORMAT_ID= '%6s';
  FORMAT_DOCUMENTO= '%11s';
  FORMAT_NOMBRE_APELLIDO= '%21s';
  FORMAT_EDAD_PESO= '%6s';
  COLUMNA_ID= 'ID';
  COLUMNA_DOCUMENTO= 'DOCUMENTO';
  COLUMNA_NOMBRE= 'NOMBRE';
  COLUMNA_APELLIDO= 'APELLIDO';
  COLUMNA_EDAD= 'EDAD';
  COLUMNA_PESO= 'PESO';
```

El estudiante tampoco hará uso de estas constantes porque ya está programado el funcionamiento del sistema que permite crear una salida estándar con el formato de tabla. Todas ellas se utilizan con ese único fin. En los **Anexos** en la página 21 se detallará la implementación, si el estudiante lo desea puede leer dicho apartado, aunque no es necesario para realizar el proyecto.

```
PROMPT= '>> ';
```

Define el `String` que representa al **prompt** del sistema. Se usará para imprimir en la salida.

```
(*Nombre de archivo de la base de datos.*)
BASEDEDATOS_NOMBRE_REAL= 'data.dat';
(*Nombre de archivo temporal de la base de datos*)
BASEDEDATOS_NOMBRE_TEMPORAL= 'tempDataBase_ka.tmpka';
```

Ambas constantes definen las cadenas de caracteres que se usarán para los archivos de base de datos que usaremos. **BASEDEDATOS_NOMBRE_REA**L se usará para utilizar el archivo de base de datos que crearemos en el sistema, y **BASEDEDATOS_NOMBRE_TEMPORAL** se usará para crear el archivo temporal usado para crear una nueva base de datos a partir de la primera y luego cambiar el nombre.

TIPOS DE DATOS

El programa utilizará varios tipos de datos para poder representar tres estructuras básicas:

1. El nombre de un comando como valor enumerado, no `String`.
2. El registro de una persona a ser guardado en un archivo.
3. El archivo mismo que se usará para guardar los datos.

TIPOS DE DATOS PARA DEFINIR EL NOMBRE DE UN COMANDO

```
TComandosSistema= (NUEVO,MODIFICAR,ELIMINAR,BUSCAR,ESTADOSIS,OPTIMIZAR,SALIR,INDEF);
```

¿Cuál es el cometido de este tipo de datos? Pues como se explicó, la unidad **UComando** no está diseñada para este sistema en particular, sino que es más amplia y puede ser usada para programar cualquier aplicación que se base en un modelo de comandos, por tanto, nuestro programa principal debe definirse los comandos que utilizará.

Cada valor de este enumerado representa a un comando particular de los 7 comandos descritos en este documento. El 8vo valor en el enumerado, llamado **INDEF**, representa al comando indefinido, el cual se puede usar para el momento en que el usuario ingresa un comando que no existe.

TIPOS DE DATOS PARA DEFINIR EL REGISTRO DE UNA PERSONA

```
{Representa el registro de una persona en el sistema.}
  TRegistroPersona= packed record
     Id: int64;
     Nombre, Apellido: String[20];
     Documento: String[10];
     Edad, Peso: byte;
     Eliminado: boolean;
  end;
```

Este tipo se usará para guardar datos en el archivo de base de datos que se creará. Puede verse que se declara como un registro empaquetado (**packed record**). Los atributos definidos en él son:

- **Id:** `int64` → Un entero grande para alojar el índice del registro dentro del archivo. Usamos el tipo numérico **int64** porque es el que retorna la operación **FileSize** que usarás a menudo en tu código, y por tanto nuestro atributo ha de ser compatible con ella.

- **Nombre** y **Apellido:** Ambos son atributos `String` de tamaño 20.

- **Documento**: Un `String` de tamaño 10 para guardar una cadena que representa el documento de la persona. Si bien en los ejemplos, tanto de este documento como de los videos, se usan siempre números, el **DOCUMENTO** de una persona puede ser cualquier cadena de caracteres.

- **Edad** y **Peso**: Dos simples atributos numéricos de tipo byte.

- **Eliminado**: Indicará si el registro ya fue eliminado o no, tal como vimos en una de las clases del curso. Un registro marcado como eliminado se ignorará por nuestro programa, pero estará aún dentro del archivo.

TIPOS DE DATOS PARA DEFINIR EL ARCHIVO DE BASE DE DATOS

```
{El archivo en el que se guardarán los datos.}
  TBaseDeDatos= file of TRegistroPersona;
```

No tiene mucha ciencia. Un simple archivo de tipo **TRegistroPersona** para guardar datos en él.

OPERACIONES YA PROGRAMADAS PARA EL ESTUDIANTE

Dado que el objetivo principal de este programa es que practiques el uso de archivos, queremos que te compliques lo menos posible con otras cuestiones, por tanto, así como te hemos provisto de la unidad **UComando** ya implementada, también te dejamos algunas operaciones ya disponibles en el programa principal para que te ayudes con ellas.

FUNCIÓN COMANDOSISTEMA

```
function comandoSistema(const c: TComando): TComandosSistema;
```

Un comando de tipo `TComando` (definido en `UComando`) tiene su nombre en formato `String` y se puede obtener mediante **nombreComando**. Sin embargo, tenerlo de esta manera requiere que el estudiante deba comparar `Strings` para ver si el nombre del comando es igual a alguna de las constantes `String` definidas con los nombres de los comandos. Si bien esto no es complicado de hacer, hará que el código de este programa quede más engorroso al tener que utilizar varias sentencias **if** para distinguir cada comando.

La función `comandoSistema` recibe una variable de tipo `TComando`, verifica su nombre internamente y luego retorna el nombre del comando como un valor enumerado de tipo `TComandoSistema`. De esta manera es muy sencillo distinguir qué comando se ingresó. Incluso, si el comando en c no es ninguno de los definidos, esta operación retornará el valor **TComandosSsitemas.INDEF**.

FUNCIÓN STRINGSEPARADORHORIZONTAL

```
function stringSeparadorHorizontal(): String;
```

Retorna simplemente una cadena de caracteres de 78 guiones. Se usará para separar el encabezado de las columnas de los valores cuando se utiliza el comando **BUSCAR**.

FUNCIÓN STRINGENCABEZADO

```
  function stringEncabezado(): String;
```

Retorna el `String` con el encabezado de las columnas de la tabla del comando **BUSCAR**. Concretamente:

ID	DOCUMENTO	NOMBRE	APELLIDO	EDAD	PESO

El estudiante utilizará esta función junto a `stringSeparadorHorizonal` para armar la cabecera de la tabla.

FUNCIÓN STRINGFILAREGISTRO

```
function stringFilaRegistro(const reg: TRegistroPersona): String;
```

Dado el registro de una persona, esta función lo retorna en formato `String` como fila de la tabla para el comando **BUSCAR**. Por ejemplo:

2	12345	Vladimir	Rodriguez	30	72

De este modo el estudiante podrá ir imprimiendo en la salida estándar uno a uno los registros de su archivo respetando el formato indicado sin complicarse en ello.

LO QUE TÚ DEBES PROGRAMAR

Recibes la unidad **UComando** ya lista, y en el programa principal tienes definidas las constantes, los tipos y hasta operaciones ya programadas para ayudarte a implementar este sistema. Tu trabajo es justamente dar funcionalidad a este sistema implementando el programa principal y una función que no está implementada y por tanto la encontrarás con código vacío:

```
{Busca en el arhivo, un registro con el documento indicado y lo asigna
a reg retornando TRUE. Si no existe en el archivo un registro con el
documento indicado entonces retorna FALSE.}
function buscarRegistro(documento: String; var reg: TRegistroPersona;
                                    var archivo: TBaseDeDatos): boolean;
```

Lo primero que debes hacer es implementar esta operación. Luego debes ir al bloque principal de tu archivo (entre **BEGIN** y **END.**) y escribir el código que permita crear un programa tal como se ha descrito la sección **El programa** en la página 5.

SUGERENCIAS

En comparación a cualquier cosa que hayas hecho en este curso, este sistema es bastante sencillo, sin embargo algunas cosas pueden tomarte por sorpresa, así que te damos aquí una muy sutiles sugerencias.

OPERACIÓN BUSCARREGISTRO

```
function buscarRegistro(documento: String; var reg: TRegistroPersona;
                                    var archivo: TBaseDeDatos): boolean;
```

Esta operación recibe como argumento el `documento` de una persona a buscar en tu archivo de base de datos, que también es recibido como argumento. Los pasos son muy simples:

1. Si el archivo está vacío se retorna **false**.
2. Si no está vacío me posiciono al principio del mismo. Mientras no llego al final del archivo:
    a. Obtengo el registro actual
    b. Comparo el documento obtenido con el parámetro **documento**.

c. Si son iguales modifico **reg** y detengo el proceso (**break**).
d. Si no son iguales avanzo una posición
e. Vuelvo al paso a

No es complejo. Lo interesante es comparar ambos documentos, que son de tipo `String`. Para ello sugerimos utilizar la operación `CompareStr` de la librería `SysUtils`:

```
function CompareStr(const S1, S2: string): Integer;
```

Como ves, recibe dos argumentos `String` como parámetros: `S1` y `S2`. Si la cadena `S1` está antes en el abecedario que `S2`, la operación retorna un valor menor que cero (0); si `S1` y S2 son iguales retorna justamente cero (0); y si `S1` está luego que `S2` en el abecedario, retorna un valor mayor que cero (0).

COMANDO ELIMINAR

El comando eliminar recibe como primer valor el parámetro **-D** o **-T**, definidos en las constantes **PARAMETRO_ELIMINAR_DOC** y **PARAMETRO_ELIMINAR_TODO** respectivamente. Puedes usar la operación **CompareStr** para obtener el primer parámetro de este comando y compararlo con estas constantes para ver cuál de ambas opciones fue ingresada, o incluso si no se corresponde con ninguna de ellas.

USO DE CONTINUE

La instrucción **continue** puede resultar muy útil, aunque no es estrictamente necesaria. Cuando se utiliza dentro de un bucle (**FOR**, **REPEAT**, **WHILE**), esta instrucción pasa por alto todas las instrucciones que están dentro del bucle siguientes a ella, y pasa a la siguiente iteración.

Por ejemplo, veamos el siguiente código:

```
a:= 0;
repeat
      readln(a);
      if a<>10 then begin
          Continue;
      end;

      writeln('Al fin ingresaste 10');
      writeln('Presiona ENTER para salir...');
      readln;

until a=10;
```

Se lee desde la entrada un entero y se guarda en **a**. El **if** de la 4ta línea verifica que **a** sea distinto de 10, y si es así se ejecuta **Continue**, lo cual implica que se volverá a ejecutar el **repeat** ignorando las tres instrucciones que hay debajo del **if**. Solo cuando **a** sea 10 el **if** no se ejecutará y se ejecutarán las tres instrucciones que hay debajo.

Esta instrucción puede facilitar mucho la implementación de detección de errores para los distintos comandos, tal como sugeriremos en el pseudocódigo más adelante.

PSEUDOCÓDIGO DEL PROGRAMA PRINCIPAL

Dado que tu trabajo, además de implementar la operación **buscarRegistro**, es programar el código principal de este programa, te dejamos a continuación una sugerencia de código:

1. Crear la base de datos o abrirla según corresponda
2. Hasta que se ingrese **SALIR**
    a. Mostrar el **prompt**
    b. Leer la entrada estándar completa
    c. Crear un comando a partir de lo leído
    d. Obtener un valor **TComandosSistema** en una variable, por ejemplo **sysCom**.
    e. **CASE sysCom OF**
        i. **NUEVO**
            1. Si no hay 5 parámetros mostrar error y continuar (`continue`).
            2. Si ya existe un registro mostrar error y continuar (`continue`).
            3. Si el parámetro `EDAD` no es numérico mostrar error (`continue`).
            4. Si el parámetro `PESO` no es numérico mostrar error (`continue`).
            5. Si no hubo error guardar datos y mostrar mensaje.
        ii. **MODIFICAR**
            1. Solución similar a `NUEVO`.
        iii. **ELIMINAR**
            1. Si hay 1 parámetro
                a. Si se ingresó **-T** borrar todo el archivo.
                b. Si no, mostrar error y *continuar*.
            2. Si hay 2 parámetros
                a. Ver si el primer parámetro es **-D**
                    i. Buscar registro
                    ii. Si hay registro marcarlo como eliminado.
                    iii. Si no hay registro mostrar mensaje.
                b. Si no es **-D** mostrar error y *continuar*.
            3. Si no hay 1 ó 2 parámetros mostrar error y *continuar*.
        iv. **BUSCAR**
            1. Si no hay parámetros imprimir todo en formato indicado.
            2. Si hay un solo parámetro
                a. Buscar registro
                    i. Si existe imprimir
                    ii. Si no, mostrar mensaje.
            3. Si hay más de 1 parámetro mostrar error y *continuar*.
        v. **ESTADOSIS**
            1. Si hay 1 o más parámetros mostrar error y *continuar*.
            2. Si no hay parámetros mostrar información.
        vi. **OPTIMIZAR**
            1. Crear nuevo archivo temporal
            2. Recorrer archivo original
                a. Si el registro actual no está eliminado
                    i. Leer registro
                    ii. Escribir en archivo temporal
                b. Si no, ignorar y avanzar.
            3. Cerrar ambos archivos.
            4. Eliminar archivo original.
            5. Cambiar nombre al archivo temporal.
            6. Abrir archivo temporal (ahora es original) y dejar abierto.
        vii. **INDEF**: Mostrar error y *continuar*.

No es estrictamente necesario hacerlo así, esto es solo una sugerencia de ayuda.

SE PIDE

Escribir un programa Pascal que cumpla con todo lo descrito en el documento y funcione exactamente igual que los ejemplos. Se te proveerá de los archivos ejecutables ya compilados para que puedas probarlos en funcionamiento y ver lo que tienes que lograr, así como de los dos archivos que componen la arquitectura de este programa:

- ✓ **`BaseDeDatosSimple.lpr`**: El programa principal que debes implementar.
- ✓ **`UComando.pas`**: Contiene las rutinas que te ayudarán a implementar todo este sistema.
- ✓ **`ENTRADA.txt`**: Contiene datos de entrada ya definidos para que tu ingreso de registros al sistema sea mucho más sencillo. Se explica su uso en la sección **Anexos**, página 21.

RESTRICCIONES DE CÓDIGO

Para este proyecto puedes crearte cualquier operación (**`function`**, **`procedure`**) que quieras, pero NO puedes definirte nuevos tipos de datos. Tampoco puedes modificar o agregar código a **`UComando`**. Debes poder resolver este proyecto con los tipos de datos y constantes definidas en el archivo principal.

No puedes utilizar ninguna facilidad de Pascal o de Lazarus que no hayamos dado en el curso. Solo puedes utilizar lo visto hasta ahora en el curso. La firma de las operaciones definidas en este documento no puede modificarse en absoluto.

ENTREGA DEL PROYECTO

Este proyecto es obligatorio y debe ser entregado ya que será evaluado por un docente. Lo que debes entregar es el archivo **`BaseDeDatosSimple.lpr`** con todo el código fuente ya incluido en él más lo que tú implementes.

Tu archivo será probado en un proyecto utilizando el archivo **`UComando.pas`** tal como se te ha provisto para este trabajo. Por tanto no debes modificar dicho archivo en absoluto ni tampoco has de modificar ninguna de las declaraciones de constantes, tipos y operaciones (subprogramas).

Dependiendo de la plataforma en que hagas el curso será el medio por el cual enviarás el archivo:

- ➢ VirtuaEdu: En la lección donde se te presenta este proyecto tendrás la opción para subir el archivo del código fuente.
- ➢ Udemy: Escribe un correo electrónico a bedelia@kaedusoft.edu.uy con el asunto **PROYECTO BaseDeDatosSimple [NOMBRE] [APELLIDO]** en el cual adjuntarás el archivo con el código fuente de tu trabajo.

Si tienes alguna duda escribe a bedelia@kaedusoft.edu.uy.

ANEXOS

En este apartado incluimos material adicional. Inicialmente explicamos cómo utilizar el archivo **`ENTRADA.txt`** para el ingreso rápido de varios registros para el sistema, de forma que puedas realizar pruebas a tu código. Luego explicamos cómo ha sido implementada la unidad **`UComando`** y las operaciones del programa principal, a modo educativo, lo cual es opcional.

USO DEL ARCHIVO ENTRADA.TXT

Siempre utilizamos la entrada estándar como forma de ingresar datos a nuestros programas, salvo cuando hemos provisto de una IGU (*GUI*) como para el proyecto ***Buscaminas*** o el ***Juego de la Vida***.

Es posible definir la entrada de un programa desde un archivo ya escrito, lo cual agiliza mucho el proceso y evita errores. Incluso es posible guardar la salida de un programa (lo que se muestra en la consola) en un archivo de texto plano.

En concreto, hemos preparado un archivo con 121 registros listos para cargar en tu archivo. ¿Cómo funciona esto? Muy simple. Para ingresar una línea en nuestro programa debemos ingresar un comando correcto y sus parámetros; por ejemplo:

```
NUEVO 111 Ana Silva 62 74
```

Con esa línea estaríamos registrando una nueva persona llamada Ana Silva con documento 111. Al presionar ENTER quedaría ingresada. Luego podemos ingresar:

```
NUEVO 112 Elena Castro 59 36
```

Y así podemos seguir y seguir. Ahora bien, podemos guardarnos un archivo TXT con las líneas de entrada ya preparadas, por ejemplo:

```
NUEVO 111 Ana Silva 62 74
NUEVO 112 Elena Castro 59 36
SALIR
```

En este caso nuestro archivo incluye el ingreso de las dos personas y además el comando SALIR para que el programa se cierre ya que mediante este proceso no podemos escribir desde la entrada estándar, sino que todas nuestras líneas de ingreso serán leídas desde el archivo. Así, nuestro programa leerá la primera línea del archivo como si se hubiese tecleado desde la entrada estándar y se hubiese presionado ENTER, luego leerá la segunda y así hasta el final.

Podemos guardar ese texto en un archivo con cualquier nombre, por ejemplo, **`ENTRADA.txt`**. Para ejecutar el programa de forma que lea la entrada desde un archivo debemos ejecutarlo desde la línea de comandos (consola o terminal) de nuestro sistema operativo.

Para abrir la terminal en ***Linux*** presiona **`CTRL+ALT+T`**. Para abrirla en ***Windows***, haz clic en el buscador y escribe **`CMD`**, cuando aparezca la opción **`símbolo del sistema`** haz clic en ella y se abrirá la consola. En este ejemplo mostraré los pasos en Windows y en Linux (son exactamente iguales):

```
Microsoft Windows [Versión 10.0.17134.765]
(c) 2018 Microsoft Corporation. Todos los derechos reservados.

C:\Users\mstrv>
```

En Linux aparecerá algo como esto:

```
kaedusoft@kaedusoft-ubuntu: ~$
```

En ambos casos debes usar el comando CD y escribir la ruta en la cual está tu archivo ejecutable y además debe estar el archivo ENTRADA.txt. En mi caso, en Windows, he puesto el archivo en el escritorio, así que escribo: **`CD C:\Users\mstrv\Desktop`**

```
C:\Users\mstrv>cd C:\Users\mstrv\Desktop

C:\Users\mstrv\Desktop>
```

En Linux es lo mismo, escribo **cd** y la ruta de mi escritorio porque allí tengo los archivos, pero si tú los tienes en otro lugar escribe esa ruta:

```
kaedusoft@kaedusoft-ubuntu: ~$ cd /home/kaedusoft/Escritorio
kaedusoft@kaedusoft-ubuntu: ~/Escritorio$
```

Ahora los pasos son idénticos, sin importar el sistema operativo. Lo que se debe hacer es escribir el nombre del programa que se quiere ejecutar, el símbolo **<** y el nombre del archivo que contiene los datos de entrada. Este procedimiento sirve para cualquier programa. Lo que se hace es indicar que lea la entrada desde donde se le indica. Se puede hacer con cualquier aplicación que tome datos desde la entrada estándar, tales como nuestros programas, `mysql` o cualquier otro. En este caso escribo:

```
BaseDeDatos < ENTRADA.txt
```

Veremos la salida del programa en la consola, como si hubiésemos escrito a mano los datos:

```
BaseDeDatosSimple < ENTRADA.txt
>> Registro agregado exitosamente
>> Registro agregado exitosamente
>> Registro agregado exitosamente
>>
```

Si quisiéramos guardar la salida de nuestro programa en un archivo de texto debemos escribir el símbolo **>** y el nombre del archivo que queremos crear con la salida (si el archivo ya existe se sobrescribirá). Por ejemplo:

```
BaseDeDatosSimple > SALIDA.txt
```

De este modo, como no ingresamos ningún archivo de entrada, debemos usar nuestro programa de forma habitual, pero no veremos la salida porque esta se guardará en el archivo que indicamos. Como se explicó, esto se puede usar para cualquier programa, incluso para los primeros que hicimos, ya que lo que el sistema operativo hace es pasarle los datos de entrada desde donde se le indican (consola o archivo) y mostrarlos donde se indica (consola o archivo).

Si queremos hacer las dos cosas a la vez simplemente escribimos ambos comandos, usando **<** para indicar la entrada de datos y **>** para indicar la salida:

```
BaseDeDatosSimple < ENTRADA.txt > SALIDA.txt
```

Esto será muy útil de ahora en más.

IMPLEMENTACIÓN DE UCOMANDO

Esta parte del documento es opcional, educativa para el estudiante. Explicamos los conceptos clave de implementación de esta unidad, los cuales no utilizan nada que no se haya visto en el curso. Todo lo que se ha usado para programar las operaciones el estudiante lo conoce y podría crear su propia unidad **UComando** con una implementación propia.

CONSTANTES Y TIPOS DE DATOS

```
unit UComando;

interface

const
MAX_PARAM_COMANDO= 25; //La cantidad máxima de parámetros
COMILLAS= #34; //Caracter de comillas dobles
```

Estas constantes ya fueron descritas anteriormente y no tienen ningún misterio ni complicación.

```
type

TParametro= record
      datoString: String;
      datoNumerico: integer;
      esNumero: boolean;
end;

TListParametros= record
      argumentos: array[1..MAX_PARAM_COMANDO] of TParametro;
      cantidad: byte;
end;

TComando= record
      nombreComando: String;
      haySiguiente: boolean;
      listaParametros: TListParametros;
      puntero: byte;
end;
```

`TParametro` y `TComando` fueron ya explicados a lo largo de este documento. Quién queda por comentar es `TListaParametros`, que solo se usa dentro de `UComando` y por tanto sus clientes no necesitan saber cómo funciona.

`TListaParametros` es simplemente un arreglo con tope, tal como hemos visto en otras oportunidades en el curso, además de haberlos utilizado en los proyectos. El arreglo contiene elementos de tipo `TParametro`, y luego se utiliza la variable `cantidad` para establecer el tope, es decir, hasta dónde se recorre el arreglo.

OPERACIONES PARA TPARAMETRO

La unidad `UComando` define una serie de operaciones para trabajar con una variable de tipo `TParametro`, y luego otra serie de operaciones para trabajar con una variable de tipo `TComando`. Veamos primero las operaciones de `TParametro`, ya que `TComando` hace uso de este tipo en sus operaciones.

FUNCIÓN ESPARAMETROSTRING

```
{Retorna TRUE si el parámetro p es de tipo numérico, FALSE si no.
Esta operación es análoga a esParametroNumerico}
function esParametroString(const p: TParametro): boolean;
begin
      result:= not p.esNumero;
end;
```

El tipo `TParametro` define un atributo booleano llamado `esNumero`, el cual debe establecerse en `TRUE` si el contenido del parámetro es un entero (solo contiene caracteres del `0` al `9`), o en `FALSE` en caso contrario. Esta operación retorna el valor opuesto para indicar si el contenido es un `String` que contiene al menos un carácter distinto de un número; por tanto, si `esNumero` es `TRUE` se retornará `FALSE` (no es un `String`), y si `esNumero` es `FALSE` se retornará `TRUE` (es un `String`).

FUNCIÓN ESPARAMETRONUMERICO

```
{Retorna TRUE si el parámetro p es de tipo numérico, FALSE si no.}
function esParametroNumerico(const p: TParametro): boolean;
begin
      result:= p.esNumero;
end;
```

Análoga a esParametroString, esta operación indica si el contenido de un parámetro contiene solo números enteros, retornando TRUE. En caso contrario retorna FALSE. Por tanto, el valor de retorno es justamente lo que haya en el atributo esNumero.

FUNCIÓN OBTENERNUMERO

```
{En caso de que p sea un parámetro de tipo numérico, asigna a n el
número contenido en p y retorna TRUE. De lo contrario, retorna FALSE.}
function obtenerNumero(var n: byte; const p: TParametro): boolean;
begin
      if esParametroNumerico(p) then begin
            n:= p.datoNumerico;
            result:= true;
      end else begin
            result:= false;
      end;
end;
```

Si el parámetro es numérico (su contenido son solo números del 0 al 9), se asigna el valor del atributo datoNumerico al argumento por referencia **n**, y retorna TRUE. En caso contrario solamente retorna FALSE y no hace nada con **n**, por tanto su valor no puede ser predicho y debe considerarse como indefinido.

FUNCIÓN OBTENERSTRING

```
{Retorna el contenido de un parámetro como un String, sin importar
si el parámetro p es numérico o no, siempre se obtiene un String.
Si, por ejemplo, el parámetro es numérico y tiene el valor 89, se
obtenddrá el String '89'.}
function obtenerString(const p: TParametro): String;
begin
      result:= p.datoString;
end;
```

Sin importar si el contenido de un parámetro es un entero puro, o bien un String cualquiera, esta operación siempre retornará ese contenido como un String, el cual estará guardado en el atributo datoString. Así pues, simplemente se retorna el valor del atributo datoString sin comprobar nada.

OPERACIONES PARA TCOMANDO

Es aquí donde todo se vuelve más interesante, ya que las operaciones de TParametro se han centrado más que nada en retornar valores, pero no procesan nada. Ahora bien, TComando es quién se encarga justamente de la creación de un comando y sus parámetros, el recorrido de los mismos y la manipulación de sus datos.

FUNCIÓN CREARCOMANDO

Esta operación es, sin duda alguna, la más importante de toda la unidad, y la más compleja, ya que es la que se encarga de recibir un String completo con una línea de entrada, y transformar eso en un comando con una lista de parámetros.

Por ejemplo, esta operación recibiría la cadena: **'CREACION "Celula Tejido" Persona 23 5689 78'** y crearía un comando con nombre **'CREACION'**, parámetro 1 **'Celula Tejido'**, parámetro 2 **'Persona'**, parámetro 3 **23**, parámetro 4 **5689** y parámetro 5 **78**. Veamos cómo funciona:

```
function crearComando(entrada: String): TComando;
var i, cantidadPalabras: byte;
  return: TComando;
  nextParam, palabra: String;
  isCompuesto: boolean;
begin
  cantidadPalabras:= WordCount(entrada,[' ']);

  return.puntero:= 0;
  return.listaParametros.cantidad:= 0;
  return.nombreComando:= ExtractWord(1,entrada,[' ']);
  isCompuesto:=false;
  for i:=2 to cantidadPalabras do begin
        palabra:= ExtractWord(i,entrada,[' ']);
        if AnsiStartsText(COMILLAS,palabra) and AnsiEndsText(COMILLAS,palabra) then begin
              nextParam:= DelChars(palabra,COMILLAS);
              isCompuesto:= false;
        end else if AnsiStartsText(COMILLAS,palabra) and not isCompuesto then begin
              isCompuesto:= true;
              nextParam:= DelChars(palabra,COMILLAS);
        end else if AnsiEndsText(COMILLAS,palabra) and isCompuesto then begin
              isCompuesto:= false;
              nextParam:= nextParam+' '+DelChars(palabra,COMILLAS);
        end else if isCompuesto then begin
              nextParam:= nextParam+' '+palabra;
        end else if not isCompuesto then begin
              nextParam:= palabra;
        end;

        if not isCompuesto then begin
              agregarParametro(nextParam,return);
        end;
  end;
  result:= return;
end;
```

Como se puede ver, esta operación solo recibe un `String` llamado entrada y retorna como resultado algo de tipo `TComando`. El `String` que se le pase como argumento deberá ser la línea leída desde la entrada estándar, así de simple. Comencemos desde donde inicia el bloque **`begin end`** de esta operación. Lo primero que tenemos es:

```
cantidadPalabras:= WordCount(entrada,[' ']);
```

Lo primero que haremos será obtener la cantidad de palabras que contiene la cadena de caracteres pasada como argumento y guardar ese valor en `cantidadPalabras`. Para ello utilizamos la operación `WordCount` de la librería `StrUtils`, la cual recibe como argumento una cadena de caracteres y un conjunto de separadores (utiliza el tipo `SET` que nosotros no hemos dado en el curso). Por ejemplo, si a esta operación se le pasa: `'Hola mundo mi primer programa'` y como separador el espacio en blanco, nos retornará como valor 5, porque hay 5 palabras separadas por espacio.

Si tuviéramos la cadena `'Hola#Mundo con un gran#dia soleado'` e invocamos a `WordCount` pasándole como separadores el espacio en blanco y el símbolo numeral o almohadilla: `[' ','#']`, tendríamos como resultado 7, ya que hay 7 palabras separadas por espacio o por el símbolo `#`.

En nuestro caso simplemente pasamos como argumento la línea de entrada y el separador de espacio en blanco para contar las palabras que hay en dicha línea de entrada. Siguiendo el ejemplo al inicio de este apartado, si en **`entrada`** está la cadena **'CREACION "Celula Tejido" Persona 23 5689 78'** , `WordCount` retornará 7, por tanto `cantidadPalabras` valdrá 7.

Nos definimos una variable local llamada `return` de tipo `TComando`, que es la que usaremos para devolver el valor al final de la operación, así que `return` es un registro con los atributos que vimos en la definición de `TComando`. A continuación, lo que se hace es inicializar los valores:

```
return.puntero:= 0;
return.listaParametros.cantidad:= 0;
return.nombreComando:= ExtractWord(1,entrada,[' ']);
```

El atributo `puntero` indicará a qué parámetro del comando se está apuntando actualmente. Como en este momento no se apunta a ninguno, y además es posible que el comando ingresado no tenga parámetros, se establece en cero (0). La cantidad de parámetros en la lista también se establece en cero (0). Finalmente, la tercera línea es quizá la más extraña, ya que hace uso de la operación `ExtractWord` de la librería `StrUtils`. La firma de esta operación es:

```
function ExtractWord(N: Integer; const S: string; const WordDelims: TSysCharSet): string;
```

Esta operación extrae una palabra concreta de una cadena de caracteres. El argumento `N` indica qué palabra se quiere extraer (1 para la primera, 2 para la segunda, etc.). El argumento `S` es la cadena desde la cual se quiere extraer la palabra. Finalmente el argumento `WordDelims` indica el conjunto de separadores a utilizar para distinguir entre palabras, tal como funciona `WordCount`. Por tanto, al invocarla de esta manera:

```
return.nombreComando:= ExtractWord(1,entrada,[' ']);
```

Se está indicando que se quiere extraer la primera palabra de la entrada que obtuvimos como argumento, teniendo en cuenta el espacio en blanco como separador. La primera palabra de la cadena será el nombre del comando, y por tanto lo guardamos en el atributo `nombreComando` de nuestra variable `return`.

Siguiendo el ejemplo anterior, si entrada es **'CREACION "Celula Tejido" Persona 23 5689 78'**, el llamado a `ExtractWord` retornará la palabra `'CREACION'`, y por tanto esa palabra se guardará en `return.nombreComando`.

Luego tenemos la línea `isCompuesto:=false;`. Esta variable es un booleano que nos permitirá saber cuándo hemos comenzado a leer un parámetro compuesto, es decir, cuando se han abierto comillas. De este modo tendremos que concatenar todas las palabras entre comillas hasta formar una única cadena que compondrá a nuestro próximo parámetro.

Ahora comienza la iteración mediante un `FOR`, el cual avanzará palabra a palabra en nuestra cadena de entrada e irá creando parámetros para añadir a la lista de parámetros del comando que estamos creando. La complicación aquí está dada por las comillas, ya que tenemos que distinguir cuando hemos comenzado a leer un parámetro compuesto y cuándo hemos terminado de hacerlo. La cabecera del `FOR` es la siguiente:

```
for i:=2 to cantidadPalabras do begin
```

Como se puede observar, iteramos desde 2 hasta la cantidad de palabras que obtuvimos al principio. Obviamente, como la primera palabra ya la usamos para el nombre del comando, no debemos contemplarla ahora, por ello iniciamos en 2. Esto además asegurad que avanzaremos solo si hay palabras disponibles, porque si la entrada ingresada no tuviera más palabras, entonces este **`FOR`** ya no iteraría porque **`i=2`** es mayor que **`cantidadPalabras=1`**.

El código interno de este `FOR` está dividido en 5 bloques `IF-THEN-ELSE` que evalúan la situación actual, y luego un `IF` final que verifica que haya que crear un nuevo parámetro. Veamos los posibles casos:

1. **La palabra actual es una palabra única encerrada entre comillas, por ejemplo:**
   `'CREACION "Tejido" 28'`

   En este caso, al extraer la cadena `'"Tejido"'`, tenemos que no es un parámetro compuesto, sino una sola palabra para la cual se han usado comillas. En este caso se extrae la cadena, se le quitan las comillas y se crea un parámetro con ella.

2. **Si la palabra empieza con comillas y no estábamos ya dentro de un parámetro compuesto:**
   Obtenemos la palabra y establecemos `isCompuesto` en `TRUE`, ya que no podemos crear un parámetro con ella porque debemos concatenar las que hay delante de ella hasta que se cierren las comillas; recién allí podremos crear el parámetro. Le quitamos las comillas y guardamos la palabra en una variable auxiliar a la que hemos llamado `nextParam`.

3. **Si la palabra termina con comillas y estamos dentro de un parámetro compuesto:**
   Establecemos `isCompuesto` en `FALSE` porque ya se ha cerrado el parámetro compuesto. Obtenemos la palabra que termina en comillas, le quitamos las comillas y la concatenamos a lo que ya teníamos en `nextParam`, agregando un espacio en blanco para separar las palabras.

4. **La palabra no inicia ni termina en comillas, pero estamos dentro de un parámetro compuesto:**
   Obtenemos la palabra y la concatenamos en `nextParam`, ya que se trata de una palabra intermedia. Por ejemplo `"Maria del Carmen"` tiene tres palabras; la palabra `'del'` no inicia ni termina con comillas, pero está dentro de un parámetro compuesto.

5. **Si no se dio ninguna de las anteriores ni estamos dentro de un parámetro compuesto:**
   Significa que estamos leyendo un parámetro común. Obtenemos la palabra y la asignamos a `nextParam` para luego crear un parámetro con ella.

Analicemos ahora la implementación de esto en Pascal, tal como está escrita en `UComando`:

```
for i:=2 to cantidadPalabras do begin
       palabra:= ExtractWord(i,entrada,[' ']);
       if AnsiStartsText(COMILLAS,palabra) and AnsiEndsText(COMILLAS,palabra) then begin
          nextParam:= DelChars(palabra,COMILLAS);
          isCompuesto:= false;
       end else if AnsiStartsText(COMILLAS,palabra) and not isCompuesto then begin
          isCompuesto:= true;
          nextParam:= DelChars(palabra,COMILLAS);
       end else if AnsiEndsText(COMILLAS,palabra) and isCompuesto then begin
          isCompuesto:= false;
          nextParam:= nextParam+' '+DelChars(palabra,COMILLAS);
       end else if isCompuesto then begin
          nextParam:= nextParam+' '+palabra;
       end else if not isCompuesto then begin
          nextParam:= palabra;
       end;

       if not isCompuesto then begin
          agregarParametro(nextParam,return);
       end;
end;
```

Lo primero que se hace en cada iteración del `FOR` es extraer la palabra actual. Vamos a ver esto en un caso concreto en que se den todas las posibles condiciones. Supongamos que en el argumento entrada se pasó la cadena de caracteres **'CREACION 12 "Tejido" Organo "Celula eucariota animal" 12 56'**.

Al leer esto, como seres humanos, nos damos cuenta enseguida de que:

- Nombre del comando: **CRAECION**
  Sus parámetros, en orden, son los siguientes:
  1. **12** → Numérico
  2. **Tejido** → No numérico
  3. **Organo** → No nuérico
  4. **Celula eucariota animal** → No numérico
  5. **12** → Numérico
  6. **56** → Numérico

Vamos a ver cómo el FOR gestiona esto. Tenemos que i va a iterar desde 2 a 9, porque hay 9 palabras separadas por espacio en esta cadena.

`I=2`, por tanto `palabra:= ExtractWord(i,entrada,[' ']);` retornará la cadena **12**, que es la segunda palabra de nuestra cadena `entrada`.

Como no se cumple ninguna de las condiciones llegamos al 5to IF de toda la cadena de `IF…THEN..ELSE`, donde se hace únicamente `nextParam:= palabra;` Así pues, `nextParam` tiene el valor `String 12`.

Saliendo de toda la cadena de `IF`, tenemos un `IF` independiente al final que verifica que no estemos dentro de un parámetro compuesto:

```
if not isCompuesto then begin
    agregarParametro(nextParam,return);
end;
```

Si no se está en un parámetro compuesto, se invoca a la operación `agregarParametro`, la cual recibe un `String` que representa al parámetro en cuestión, y una variable por referencia de tipo `TComando`. En esta invocación se pasa como primer argumento a `nextParam` (que en este caso es `12`) y como segundo argumento por referencia a `return`, nuestra variable local de tipo `TComando` que usaremos para devolver el retorno de `crearComando`. La operación `agregarParametro` será la encargada de crear un parámetro de tipo `TParametro` con los datos pasados y agregarlo al final de la lista de parámetros. La veremos en detalle luego de terminar este análisis.

Así pues, se ha agregado el primer parámetro a la lista de parámetros del comando.

`I=3`, por tanto `palabra:= ExtractWord(i,entrada,[' ']);` retornará la cadena **"Tedjido"** (incluyendo las comillas), que es la tercera palabra de nuestra cadena `entrada`.

Entramos en el primer `IF`:

```
if AnsiStartsText(COMILLAS,palabra) and AnsiEndsText(COMILLAS,palabra)
then begin
      nextParam:= DelChars(palabra,COMILLAS);
      isCompuesto:= false;
end else…
```

En las condiciones del IF podemos ver la operación `AnsiStartsText` de la librería `StrUtils`. Esta operación recibe dos `Strings` como argumento: el primero indica un subtexto y el segundo un texto. Retornará `TRUE` si el texto empieza con el subtexto. Por ejemplo: `'TOR'` es el subtexto y `'TORMENTA'` el texto, pues resulta que `'TORMENTA'` empieza con `'TOR'` así que retorna `TRUE`. En caso contrario retorna `FALSE`. No verifica mayúsculas y minúsculas.

La operación AnsiEndsText es igual que AnsiStartsText solo que verifica el final de la cadena.

Por tanto, lo que hacemos en la primera condición es verificar que la palabra leída empiece con comillas, y en la segunda que termine con comillas. Como las une un AND, si ambas condiciones se cumplen, estamos con una palabra que empieza y termina con comillas, como es el caso, pero no es un parámetro compuesto.

```
nextParam:= DelChars(palabra,COMILLAS);
```

La operación DelChars de StrUtils, recibe un String como argumento y un carácter como segundo argumento. Borra del primer argumento todas las ocurrencias del carácter pasado. Por ejemplo, si el primer argumento es 'TORMENTA' y se le pasa como segundo argumento 'T', retornará un nuevo String sin las 'T' en él: 'ORMENA'.

En este caso, lo que hacemos es quitar las comillas, por tanto nextParam queda con el valor **Tejido**. Luego de esto se establece isCompuesto en FALSE para indicar que no estamos en un parámetro compuesto, y pasamos al IF del final, que justamente verifica que isCompuesto sea FALSE. Agregamos este parámetro a la lista de parámetros mediante agregarParametro y volvemos a iterar.

I=4, por tanto `palabra:= ExtractWord(i,entrada,[' ']);` retornará la cadena **Organo** que es la cuarta palabra de nuestra cadena entrada.

Pasamos al quinto IF, tal como con el parámetro **12**. Así pues se agregará Organo a la lista.

I=6, por tanto `palabra:= ExtractWord(i,entrada,[' ']);` retornará la cadena **"Celula** (incluyendo las comillas), que es la sexta palabra de nuestra cadena entrada.

Ahora tenemos una palabra que empieza con comillas, lo cual indica que leeremos un parámetro compuesto, por tanto no podemos agregar **Celula** como parámetro hasta haber leído todas las palabras encerradas en comillas. Pasamos así al segundo IF de nuestro FOR:

```
…end else if AnsiStartsText(COMILLAS,palabra) and not isCompuesto then
begin
      isCompuesto:= true;
      nextParam:= DelChars(palabra,COMILLAS);
end else…
```

Su condición verifica que la palabra empiece con comillas y que no estemos ya en u parámetro compuesto. Como ambas se cumplen, ahora sí establecemos isCompuesto en TRUE. Asignamos a nextParam la palabra leída quitándole previamente las comillas con DelChars. Así que nextParam queda con el valor **Celula**.

Al llegar al IF final, este no se ejecutará, porque su condición es que no estemos dentro de un parámetro compuesto, y pues, como sí lo estamos, no se agregará nada nuevo todavía a la lista de parámetros.

I=7, por tanto `palabra:= ExtractWord(i,entrada,[' ']);` retornará la cadena **eucariota**, que es la séptima palabra de nuestra cadena entrada.

Entramos por tanto en el cuarto IF, ya que hemos leído una palabra única, pero isCompuesto es TRUE:

```
end else if isCompuesto then begin
      nextParam:= nextParam+' '+palabra;
```

Lo que hace este IF simplemente es concatenar lo que ya había en nextParam, que es Celula, un espacio, y la palabra leída. Por tanto ahora nextParam tiene la cadena Celula eucariota.

El último IF no se ejecutará porque isCompuesto aun es TRUE, por tanto no se agrega un nuevo parámetro.

I=8, por tanto `palabra:= ExtractWord(i,entrada,[' ']);` retornará la cadena **animal"** (incluyendo las comillas), que es la octava palabra de nuestra cadena entrada.

Entraremos así en el tercer IF de nuestro FOR, que justamente evalúa que tengamos una palabra finalizada en comillas para indicar que el parámetro compuesto ya se ha leído todo:

```
end else if AnsiEndsText(COMILLAS,palabra) and isCompuesto then begin
      isCompuesto:= false;
      nextParam:= nextParam+' '+DelChars(palabra,COMILLAS);
```

Se indica que isCompuesto sea FALSE y se añade a nextParam la última palabra leída, concatenándola con lo anterior y un espacio entre medio. Queda nexParam con el valor **Celula eucariota animal**.

El último IF ahora sí se ejecutará, y añadirá el parámetro con el valor **Celula eucariota animal**.

Finalmente el FOR continuará hasta terminar la cadena, leyendo los dos últimos parámetros que quedan pendientes.

Te recomendamos que uses **UComando** para crear un programa simple que lea una entrada desde la entrada estándar e invoques a crearComando usando el depurador, así puedes ver paso por paso cómo funciona toda esta mecánica. Es la forma más adecuada de estudiar este código.

PROCEDIMIENTO AGREGARPARAMETRO

```
{Esta operación es privada y no es accesible desde fuera de la unidad.
Agrega un nuevo parámetro a la lista de parámetros del comando c.}
procedure agregarParametro(value: String; var c: TComando);
var newParam: TParametro;

begin
      if TryStrToInt(value,newParam.datoNumerico) then begin
          newParam.datoNumerico:= StrToInt(value);
          newParam.esNumero:= true;
          newParam.datoString:= value;
      end else begin
          newParam.esNumero:= false;
          newParam.datoString:= value;
      end;

      c.listaParametros.cantidad+= 1;
      c.listaParametros.argumentos[c.listaParametros.cantidad]:= newParam;

end;
```

Recibe un String, crea un parámetro TParametro y lo agrega a la lista de parámetros de **c**, que es un argumento por referencia de tipo TComando.

Definimos una variable local de tipo `TParametro` llamada `newParam`, en la cual guardaremos los datos y luego la agregaremos a la lista.

Lo primero que hacemos es utilizar la operación `TryStrToInt` de la librería `SysUtils`. Esta operación intenta obtener el valor numérico de un `String`, guardarlo en un argumento por referencia y retornar `TRUE` si se pudo lograr. En caso de que el `String` no fuera un número entonces se retorna `FALSE`.

Si `TryStrToInt` funciona se inicializa `newParam` como parámetro numérico. Incluso el atributo `datoString` recibe el valor `String` del número. Si no funciona, entonces se inicializa `newParam` como no numérico.

Las dos líneas:

```
c.listaParametros.cantidad+= 1;
c.listaParametros.argumentos[c.listaParametros.cantidad]:= newParam;
```

Aumentan la cantidad de parámetros en la lista, y luego asignan el valor `newParam` a la celda tope de dicha lista.

FUNCIÓN NOMBRECOMANDO

```
function nombreComando(const c: TComando): String;
begin
	result:= c.nombreComando;
end;
```

Simplemente retorna el valor del atributo **`nombreComando`** del argumento **`c`** pasado. Es muy simple.

FUNCIÓN HAYSIGUIENTEPARAMETRO

```
function haySiguienteParametro(const c: TComando): boolean;
begin
	result:= c.puntero<cantidadParametros(c);
end;
```

Verifica que el valor del atributo **`puntero`** sea menor a la cantidad de parámetros en la lista de parámetros de **`c`**. Si el puntero es menor significa que aún hay uno o más parámetros que ver, en caso contrario ya se han visitado todos los parámetros de la lista. Retorna `TRUE` o `FALSE` respectivamente.

FUNCIÓN SIGUIENTEPARAMETRO

```
function siguienteParametro(var c: TComando): TParametro;
begin
	if haySiguienteParametro(c) then begin
		c.puntero+= 1;
		result:= c.listaParametros.argumentos[c.puntero];
	end else begin
		result.datoString:= 'NULL';
		result.esNumero:= false;
	end;
end;
```

Si aún hay parámetros por ver (`haySiguienteParametro`), se aumenta en `1` el valor del atributo `puntero`, y luego se obtiene el parámetro ubicado en esa posición de la lista.

Si no hay parámetros por ver, se retorna un parámetro con el valor `NULL`.

FUNCIÓN RESETEARPARAMETROS

```
procedure resetearParametros(var c: TComando);
begin
      c.puntero:= 0;
end;
```

Simplemente establece nuevamente el valor del atributo `puntero` en cero (0). Con eso, las operaciones `haySiguienteParametro` y `siguienteParametro` volverán a recorrer la lista desde el principio.

FUNCIÓN CANTIDADPARAMETROS

```
function cantidadParametros(const c: TComando): byte;
begin
      result:= c.listaParametros.cantidad;
end;
```

Retorna el valor del tope de la lista de parámetros, representado por el atributo **`cantidad`**.