```
#ifndef COMANDOS_H
#define COMANDOS_H
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <stddef.h>
#include <string.h>
#include "lista.h"
/* ******************************************************************
 *                    DEFINICION DE LOS TIPOS DE DATOS
 * *****************************************************************/

typedef struct sesion sesion_t;
typedef struct curso curso_t;
typedef struct base base_t;

// Inicializa la base de datos.
// Pre: ninguna.
// Post: Se inicializo la base de datos necesaria para todo el programa.
base_t* inicializar_base_de_datos();

// Se cierra adecuadamente la base de datos.
// Pre: La base de datos fue creada.
// Post: La base de datos fue cerrada adecuadamente.
void cerrar_base_de_datos(base_t* basedatos);

// Crea una sesion.
// Pre: Ninguna.
// Post: se creo una sesion
sesion_t* sesion_crear();

// Destruye una sesion
```

```
    // Pre: la sesion fue creada.
34  // Post: se destruyo la sesion.
    void sesion_destruir(sesion_t* sesion);
36
    // Obtiene el padron de la sesion en curso.
38  // Pre: La sesion fue creada.
    // Post: Devuelve el padron de la sesion en curso
40  char* obtener_padron(sesion_t* sesion);

42  /* ******************************************************
     *                    COMANDOS DEL PROGRAMA
44   * ******************************************************/

46   /* Primitivas de administracion */

48  // Agrega un nuevo curso a la base de datos.
    // Pre: La base de datos fue creada.
50  // Post: Devuelve 0 si se ejecuto correctamente, -1 si el curso ya existia, -2 s
    i no se pudo agregar.
    int agregar_curso(char* idc,char* descripcion,char* materia,char* vacantes, base
    _t* basedatos);
52
    // Inscribe un alumno en un curso.
54  // Pre: El curso se encuentra en la base de datos, la cual fue creada.
    // Post: Devuelve 1 si el alumno quedo como regular, 2 si quedo como condicional
     y
56  // -1 si el alumno ya estaba inscripto
    int inscribir(char* padron, char* idc, base_t* basedatos);
58
    // Elimina un curso de la base de datos.
60  // Pre: La base de datos fue creada.
    // Post: Devuelve 0 si se elimino el curso, -1 si el curso no existe.
```

```
62  int eliminar_curso(char* idc,base_t* basedatos);

64  // Elimina un alumno de un curso.
    // Pre: La base de datos fue creada.
66  // Post: Devuelve 0 si desinscribio al alumno, −1 si el curso no existe y
    // −2 si el alumno no estaba inscripto
68  int desinscribir(char* padron, char* idc, base_t* basedatos);

70  // Muestra un listado de los inscriptos al curso.
    // Pre: La base de datos fue creada.
72  // Post: Devuelve 0 si no hubo problema, −1 si el curso no existe.
    int listar_inscriptos(char* idc, base_t* basedatos);
74
    /* Primitivas para alumnos */
76
    // Muestra una lista de los cursos filtrando por materia de ser especificada.
78  // Pre: La base de datos fue creada.
    // Post: Se listan todos los cursos o los de la materia especificada.
80  void listar_cursos(char* filtro,base_t* basedatos);

82  // Inicia una nueva sesion asociada a un padron
    // Pre: La sesion fue creada.
84  // Post: Devuelve 0 si inicio la sesion exitosamente, −1 si ya habia una
    // sesion abierta.
86  int sesion_iniciar(sesion_t* sesion,char* padron);

88  // Selecciona un curso para inscribirse al finalizar la sesion
    // Pre: La base de datos y la sesion fueron creadas.
90  // Post: Se agrega el curso a la lista de inscripciones para aplicar luego.
    // Devuelve 1 si se inscribio como regular, 2 si fue como condicional,
92  // −1 si ya estaba inscripto en el curso, −2 si no existe el curso, −3 si no
    // hay sesion abierta, −4 si ya estaba pendiente la inscripcion, −5 si hubo un e
```

```
rror.
int sesion_inscribir(char* idc,base_t* basedatos, sesion_t* sesion);

// Muestra el estado actual de la sesion y los cursos seleccionados en sesion_in
scribir
// Pre: La sesion fue creada.
// Post: Muestra el estado actual de la sesion y los cursos a inscribirse de hab
erlos.
void sesion_ver(sesion_t* sesion);

// Des-selecciona el ultimo curso seleccionado con sesion_inscribir. Puede ser
// utilizado varias veces seguidas.
// Pre: La base de datos y la sesion fueron creadas.
// Post: Devuelve 0 si pudo quitar la ultima inscripcion, -1 si no hay sesion en
// curso, -2 si no hay acciones para deshacer.
int sesion_deshacer(sesion_t* sesion);

// Inscribe al alumno en los cursos seleccionados y cierra la sesion.
// Pre: La base de datos y la sesion fueron creadas.
// Post: Devuelve 0 si fue exitoso, -1 si no hay sesion iniciada.
int sesion_aplicar(sesion_t* sesion,base_t* basedatos);

#endif // COMANDOS_H
```

```
#ifndef LISTA_H
#define LISTA_H
#include <stdbool.h>
#include <stddef.h>

/* ******************************************************************
 *                    DEFINICION DE LOS TIPOS DE DATOS
 * ******************************************************************/

/* La lista estÃ¡ planteada como una lista de punteros genÃ©ricos. */

typedef struct lista lista_t;
typedef struct lista_iter lista_iter_t;

/* ******************************************************************
 *                        PRIMITIVAS DE LA LISTA
 * ******************************************************************/


/* Primitivas basicas */

// Crea una lista.
// Post: devuelve una nueva lista vacÃ-a.
lista_t *lista_crear();

// Devuelve verdadero o falso, segÃºn si la lista tiene o no elementos.
// Pre: la lista fue creada.
bool lista_esta_vacia(const lista_t *lista);

// Inserta un elemento al principio de la lista. Devuelve verdadero si pudo inse
rtarlo.
// Pre: la lista fue creada.
```

```
32  // Post: se agrega el elemento al principio de la lista.
    bool lista_insertar_primero(lista_t *lista, void *dato);
34
    // Inserta un elemento al final de la lista. Devuelve verdadero si pudo insertar
    lo.
36  // Pre: la lista fue creada.
    // Post: se agrega el elemento al final de la lista.
38  bool lista_insertar_ultimo(lista_t *lista, void *dato);

40  // Saca el primer elemento de la lista. Si la lista tiene elementos, se quita el
    // primero de la lista, y se devuelve su valor, si estÃ¡ vacÃ-a, devuelve NULL.
42  // Pre: la lista fue creada.
    // Post: se devolviÃ³ el valor del primer elemento anterior, la lista
44  // contiene un elemento menos, si la lista no estaba vacÃ-a.
    void *lista_borrar_primero(lista_t *lista);
46
    // Obtiene el valor del primer elemento de la lista. Si la lista tiene
48  // elementos, se devuelve el valor del primero, si estÃ¡ vacÃ-a devuelve NULL.
    // Pre: la lista fue creada.
50  // Post: se devolviÃ³ el primer elemento de la lista, cuando no estÃ¡ vacÃ-a.
    void *lista_ver_primero(const lista_t *lista);
52
    // Devuelve la cantidad de elementos que hay en la lista.
54  // Pre: La lista fue creada.
    // Post: Se devuelve la cantidad de elementos que hay listados.
56  size_t lista_largo(const lista_t *lista);

58  // Destruye la lista. Si se recibe la funciÃ³n destruir_dato por parÃ¡metro,
    // para cada uno de los elementos de la lista llama a destruir_dato.
60  // Pre: la lista fue creada. destruir_dato es una funciÃ³n capaz de destruir
    // los datos de la lista, o NULL en caso de que no se la utilice.
62  // Post: se eliminaron todos los elementos de la lista.
```

```
    void lista_destruir(lista_t *lista, void destruir_dato(void *));
64
    /* Primitivas de iterador externo */
66
    // Crea un iterador externo asociado a una lista.
68  // Pre: La lista fue creada.
    // Post: Se devuelve un iterador apuntando al primer elemento de la lista.
70  // Si la lista estuviese vacia, se devuelve NULL.
    lista_iter_t *lista_iter_crear(const lista_t *lista);
72
    // Mueve el iterador al siguiente elemento de la lista.
74  // Pre: El iterador y la lista correspondiente fueron creados.
    // Post: Se devuelve true y el iterador apunta al siguiente elemento
76  // o NULL en caso de estar en el final de la lista.
    bool lista_iter_avanzar(lista_iter_t *iter);
78
    // Devuelve el valor al que esta apuntando el iterador.
80  // Pre: El iterador y la lista correspondiente fueron creados.
    // Post: Se devuelve el dato al que se encontraba apuntando el iterador.
82  void *lista_iter_ver_actual(const lista_iter_t *iter);

84  // Verifica si el iterador esta al final de la lista.
    // Pre: El iterador y la lista correspondiente fueron creados.
86  // Post: Devuelve true si el iterador esta al final, false en caso contrario.
    bool lista_iter_al_final(const lista_iter_t *iter);
88
    // Destruye el iterador.
90  // Pre: El iterador fue creado.
    // Post: Se libera la memoria del iterador.
92  void lista_iter_destruir(lista_iter_t *iter);

94  /* Primitivas de lista junto con iterador externo */
```

```
96  // Se inserta un elemento en la posicion que apunta el iterador.
    // Pre: El iterador y la lista correspondiente fueron creados.
98  // Post: Se agrego el elemento en la posicion apuntada y quedo el iterador apuntandolo.
    bool lista_insertar(lista_t *lista, lista_iter_t *iter, void *dato);
100
    // Se devuelve y deslista el elemento al que apunta el iterador.
102 // Pre: El iterador y la lista correspondiente fueron creados.
    // Post: Se devuelve el dato al que apuntaba el iterador luego de sacarlo de la lista.
104 void *lista_borrar(lista_t *lista, lista_iter_t *iter);

106 /* Primitivas de iterador interno */

108 // Ejecuta la funcion visitar a todos los elementos de la lista, el parametro extra es para esta funcion.
    // Pre: La lista fue creada. La funcion visitar es acorde a los elementos de la lista.
110 // Post: Se procesaron con visitar todos los elementos de la lista.
    void lista_iterar(lista_t *lista, bool (*visitar)(void *dato, void *extra), void *extra);
112
    #endif // LISTA_H
```

```
#ifndef COMANDOS_C
#define COMANDOS_C
#include "lista.h"
#include "comandos.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <stddef.h>
#include <string.h>
#define LIDC 11
#define LDESCRIPCION 81
#define LMATERIA 11
#define LVACANTES 4
#define LPADRON 6

typedef struct inscripcion{
    char* curso;
    bool estado;
}inscripcion_t;

struct sesion{
    char* padron;
    lista_t* inscripciones;
};

struct curso{
    char* idc;
    char* descripcion;
    char* materia;
    int vacantes;
    lista_t* inscriptos;
};
```

```
34  struct base{
        lista_t* listadecursos;
36  };

38  typedef bool(*funcioniterador)(void*, void*);
    typedef void (*destruccion_dato)(void*);
40
    void destruir_inscripto(char* padron){
42      free(padron);
    }
44
    void destruir_inscripcion(inscripcion_t* inscripcion){
46      free(inscripcion->curso);
        free(inscripcion);
48  }

50  curso_t* crear_curso(char* idc,char* descripcion,char* materia,char* vacantes){
        curso_t* nuevocurso = calloc(1,sizeof(curso_t));
52      if (nuevocurso==NULL) return NULL;
        nuevocurso->inscriptos = lista_crear();
54      if (nuevocurso->inscriptos==NULL){
            free(nuevocurso);
56          return NULL;
        }
58      nuevocurso->idc = malloc(LIDC*sizeof(char));
        nuevocurso->descripcion = malloc(LDESCRIPCION*sizeof(char));
60      nuevocurso->materia = malloc(LMATERIA*sizeof(char));
        if ((nuevocurso->idc==NULL) || (nuevocurso->descripcion==NULL) || (nuevocurs
    o->materia==NULL)){
62          if (nuevocurso->idc==NULL) free(nuevocurso->idc);
            if (nuevocurso->descripcion==NULL) free(nuevocurso->descripcion);
```

```
64          if (nuevocurso->materia==NULL) free(nuevocurso->materia);
            free(nuevocurso);
66      }
        strcpy(nuevocurso->idc,idc);
68      strcpy(nuevocurso->descripcion,descripcion);
        strcpy(nuevocurso->materia,materia);
70      nuevocurso->vacantes = atoi(vacantes);
        return nuevocurso;
72  }

74  void destruir_curso(curso_t* curso){
        free(curso->idc);
76      free(curso->descripcion);
        free(curso->materia);
78      lista_destruir(curso->inscriptos,(destruccion_dato)&destruir_inscripto);
        free(curso);
80  }

82  base_t* inicializar_base_de_datos(){
        base_t* basededatos = malloc(sizeof(base_t));
84      if (basededatos==NULL) return NULL;
        basededatos->listadecursos = lista_crear();
86      if (basededatos->listadecursos==NULL){
            free(basededatos);
88          return NULL;
        }
90      return basededatos;
    }
92
    void cerrar_base_de_datos(base_t* basededatos){
94      lista_destruir(basededatos->listadecursos,(destruccion_dato)&destruir_curso)
    ;
```

```
        free(basededatos);
96  }

98  int existe_curso(char* idc,lista_t* lista){
        int vuelta = -1;
100     int i = 0;
        lista_iter_t* iterador = lista_iter_crear(lista);
102     curso_t* curso;
        do{
104         curso = (curso_t*) lista_iter_ver_actual(iterador);
            if (curso!=NULL){
106             if (strcmp(idc,curso->idc)==0) vuelta=i;
            }
108         i++;
        }while(lista_iter_avanzar(iterador) && (vuelta==-1));
110     lista_iter_destruir(iterador);
        return vuelta;
112 }

114 int alumno_inscripto(char* padron,lista_t* lista){
        int i = 0;
116     int estainscripto = -1;
        char* inscripto;
118     lista_iter_t* iterador = lista_iter_crear(lista);
        while((!lista_iter_al_final(iterador)) && (estainscripto==-1)){
120         inscripto = (char*) lista_iter_ver_actual(iterador);
            if (strcmp(inscripto,padron)==0) estainscripto=i;
122         lista_iter_avanzar(iterador);
            i++;
124     }
        lista_iter_destruir(iterador);
126     return estainscripto;
```

```
    }
128
    curso_t* conseguir_curso(char* idc, lista_t* lista){
130     int i;
        int posicion = existe_curso(idc,lista);
132     lista_iter_t* iterador = lista_iter_crear(lista);
        for(i=0;i<posicion;i++) lista_iter_avanzar(iterador);
134     curso_t* curso = NULL;
        curso = lista_iter_ver_actual(iterador);
136     lista_iter_destruir(iterador);
        return curso;
138 }

140 bool imprimir_inscripto(char* padron, int* estado){
        fprintf(stdout,"%s ",padron);
142     (*estado>0) ? fprintf(stdout,"regular\n") : fprintf(stdout,"condicional\n");
        (*estado)--;
144     return true;
    }
146
    int agregar_curso(char* idc,char* descripcion,char* materia,char* vacantes, base
    _t* basedatos){
148     if (existe_curso(idc,basedatos->listadecursos)!=-1) return -1;
        curso_t* curso = crear_curso(idc,descripcion,materia,vacantes);
150     lista_insertar_ultimo(basedatos->listadecursos,curso);
        return 0;
152 }

154 int inscribir(char* padron, char* idc, base_t* basedatos){
        int pos = existe_curso(idc,basedatos->listadecursos);
156     if (pos==-1) return -2;
        char* anotar = malloc(LPADRON*sizeof(char));
```

```
158         strcpy(anotar,padron);
            curso_t* curso = conseguir_curso(idc,basedatos->listadecursos);
160         if (alumno_inscripto(anotar,curso->inscriptos)!=-1){
                free(anotar);
162             return -1;
            }
164         lista_insertar_ultimo(curso->inscriptos,anotar);
            if (curso->vacantes<lista_largo(curso->inscriptos)) return 2;
166         return 1;
    }
168
    int eliminar_curso(char* idc,base_t* basedatos){
170         int i;
            int pos = existe_curso(idc,basedatos->listadecursos);
172         if (pos==-1) return -1;
            lista_iter_t* iterador = lista_iter_crear(basedatos->listadecursos);
174         for(i=0;i<pos;i++) lista_iter_avanzar(iterador);
            destruir_curso((curso_t*)lista_borrar(basedatos->listadecursos,iterador));
176         lista_iter_destruir(iterador);
            return 0;
178 }

180 int desinscribir(char* padron, char* idc, base_t* basedatos){
            int i,posicion;
182         curso_t* curso = conseguir_curso(idc,basedatos->listadecursos);
            if (curso==NULL) return -1;
184         posicion = alumno_inscripto(padron,curso->inscriptos);
            if (posicion==-1) return -2;
186         lista_iter_t* iterador = lista_iter_crear(curso->inscriptos);
            for(i=0;i<posicion;i++) lista_iter_avanzar(iterador);
188         destruir_inscripto((char*)lista_borrar(curso->inscriptos,iterador));
            lista_iter_destruir(iterador);
```

```
190     return 0;
    }
192
    int listar_inscriptos(char* idc, base_t* basedatos){
194     int pos = existe_curso(idc,basedatos->listadecursos);
        if (pos==-1) return -1;
196     curso_t* curso = conseguir_curso(idc,basedatos->listadecursos);
        int regularocondicional = curso->vacantes;
198     lista_iterar(curso->inscriptos,(funcioniterador)&imprimir_inscripto,(void*)&
    regularocondicional);
        return 0;
200 }

202 void listar_cursos(char* filtro,base_t* basedatos){
        lista_iter_t* iterador = lista_iter_crear(basedatos->listadecursos);
204     curso_t* curso;
        int inscriptos;
206     while(!lista_iter_al_final(iterador)){
            curso = lista_iter_ver_actual(iterador);
208         inscriptos = lista_largo(curso->inscriptos);
            if (filtro[0]=='\0'){
210             fprintf(stdout,"%s: %s (%s) Vacantes: %d Inscriptos: %d\n",curso->idc,curso->de
    scripcion,curso->materia,curso->vacantes,inscriptos);
            }else{
212             if (strcmp(filtro,curso->materia)==0) fprintf(stdout,"%s: %s (%s) Vacant
    es: %d Inscriptos: %d\n",curso->idc,curso->descripcion,curso->materia,curso->vacantes,i
    nscriptos);
            }
214         lista_iter_avanzar(iterador);
        }
216     lista_iter_destruir(iterador);
    }
```

```
218
    sesion_t* sesion_crear(){
220     sesion_t* sesion = calloc(1,sizeof(sesion_t));
        if (sesion==NULL) return NULL;
222     sesion->padron = calloc(LPADRON+1,sizeof(char));
        sesion->inscripciones = lista_crear();
224     if (sesion->inscripciones== NULL || sesion->padron==NULL){
            if (sesion->inscripciones!=NULL) free(sesion->inscripciones);
226         if (sesion->padron!=NULL) free(sesion->padron);
            free(sesion);
228         return NULL;
        }
230     strcpy(sesion->padron,"00000");
        return sesion;
232 }

234 void sesion_destruir(sesion_t* sesion){
        lista_destruir(sesion->inscripciones,(destruccion_dato)&destruir_inscripcion
    );
236     free(sesion->padron);
        free(sesion);
238 }

240 int sesion_iniciar(sesion_t* sesion,char* padron){
        if (strcmp(sesion->padron,"00000")!=0) return -1;
242     strcpy(sesion->padron,padron);
        return 0;
244 }

246 char* obtener_padron(sesion_t* sesion){
        char* padron = malloc(LPADRON*sizeof(char));
248     strcpy(padron,sesion->padron);
```

```
        return padron;
250 }

252 int sesion_inscribir(char* idc,base_t* basedatos, sesion_t* sesion){
        //verifico si hay sesion iniciada
254     if (strcmp(sesion->padron,"00000")==0) return -3;
        //verifico si existe el curso
256     int pos = existe_curso(idc,basedatos->listadecursos);
        if (pos==-1) return -2;
258     //verifico si ya estaba programada la inscripcion
        bool yaanotado = false;
260     inscripcion_t* inscripcion;
        lista_iter_t* iterador = lista_iter_crear(sesion->inscripciones);
262     while(!lista_iter_al_final(iterador) && !yaanotado){
            inscripcion = (inscripcion_t*)lista_iter_ver_actual(iterador);
264         if (strcmp(inscripcion->curso,idc)==0) yaanotado=true;
            lista_iter_avanzar(iterador);
266     }
        lista_iter_destruir(iterador);
268     if (yaanotado==true) return -4;
        //consigo el curso
270     int i;
        iterador = lista_iter_crear(basedatos->listadecursos);
272     for(i=0;i<pos;i++) lista_iter_avanzar(iterador);
        curso_t* curso = (curso_t*) lista_iter_ver_actual(iterador);
274     lista_iter_destruir(iterador);
        //verifico si ya esta anotado en el curso
276     if (alumno_inscripto(sesion->padron,curso->inscriptos)!=-1) return -1;
        //creo la inscripcion
278     inscripcion = malloc(sizeof(inscripcion_t));
        if (inscripcion==NULL) return -5;
280     //programo la inscripcion
```

```
        inscripcion->curso = malloc(LIDC*sizeof(char));
282     strcpy(inscripcion->curso,idc);
        ((curso->vacantes)>lista_largo(curso->inscriptos)) ? (inscripcion->estado=fa
    lse) : (inscripcion->estado=true);
284     lista_insertar_ultimo(sesion->inscripciones,inscripcion);
        if (inscripcion->estado) return 1;
286     return 2;
    }
288
    void sesion_ver(sesion_t* sesion){
290     if (strcmp(sesion->padron,"00000")==0){
            fprintf(stdout,"Error: no hay una sesion en curso\n");
292         return;
        }
294     fprintf(stdout,"Padron: %s\n",sesion->padron);
        if (lista_largo(sesion->inscripciones)==0){
296         fprintf(stdout,"No hay inscripciones\n");
            return;
298     }
        lista_iter_t* iterador = lista_iter_crear(sesion->inscripciones);
300     inscripcion_t* inscripcion;
        while(!lista_iter_al_final(iterador)){
302         inscripcion = (inscripcion_t*)lista_iter_ver_actual(iterador);
            fprintf(stdout,"%s ",inscripcion->curso);
304         (inscripcion->estado) ? fprintf(stdout,"(condicional)\n") : fprintf(stdout,"(
    regular)\n");
            lista_iter_avanzar(iterador);
306     }
        lista_iter_destruir(iterador);
308 }

310 int sesion_deshacer(sesion_t* sesion){
```

```
        if (strcmp(sesion->padron,"00000")==0) return -1;
312     if (lista_largo(sesion->inscripciones)==0) return -2;
        lista_iter_t* iteradordeprueba = lista_iter_crear(sesion->inscripciones);
314     lista_iter_t* iteradorreal = lista_iter_crear(sesion->inscripciones);
        while(!lista_iter_al_final(iteradordeprueba)){
316         lista_iter_avanzar(iteradordeprueba);
            if (!lista_iter_al_final(iteradordeprueba)) lista_iter_avanzar(iteradorr
    eal);
318     }
        destruir_inscripcion((inscripcion_t*)lista_borrar(sesion->inscripciones,iter
    adorreal));
320     //destruir_inscripcion((inscripcion_t*)lista_borrar_primero(sesion->inscripc
    iones));
        lista_iter_destruir(iteradordeprueba);
322     lista_iter_destruir(iteradorreal);
        return 0;
324 }

326 int sesion_aplicar(sesion_t* sesion,base_t* basedatos){
        inscripcion_t* anotado;
328     if (strcmp(sesion->padron,"00000")==0) return -1;
        while(!lista_esta_vacia(sesion->inscripciones)){
330         anotado = (inscripcion_t*)lista_borrar_primero(sesion->inscripciones);
            inscribir(sesion->padron,anotado->curso,basedatos);
332         destruir_inscripcion(anotado);
        }
334     strcpy(sesion->padron,"00000");
        return 0;
336 }

338 #endif // COMANDOS_C
```

```c
#ifndef LISTA_C
#define LISTA_C
#include "lista.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <stddef.h>

/* ******************************************************************
 *                    DEFINICION DE LOS TIPOS DE DATOS
 * *****************************************************************/

/* Se trata de una cola que contiene datos de tipo void*
 * (punteros genÃ©ricos). */

typedef struct nodo_lista {
    void* datos;
    struct nodo_lista* siguiente;
}nodo_lista_t;

struct lista {
    struct nodo_lista* primero;
    struct nodo_lista* ultimo;
    size_t cantidad;
};

struct lista_iter {
    struct nodo_lista* actual;
    struct nodo_lista* anterior;
};

/* ******************************************************************
```

```
 *                          PRIMITIVAS DE LA LISTA
 * ********************************************************************/

lista_t *lista_crear(){
    lista_t* lista = malloc(sizeof(lista_t));
    if (lista==NULL) return NULL;
    lista->primero=NULL;
    lista->ultimo=NULL;
    lista->cantidad=0;
    return lista;
}

bool lista_esta_vacia(const lista_t *lista){
    return (lista->primero==NULL);
}

bool lista_insertar_primero(lista_t *lista, void *dato){
    nodo_lista_t* nuevonodo = malloc(sizeof(nodo_lista_t));
    if (nuevonodo==NULL) return false;
    nuevonodo->datos = dato;
    nuevonodo->siguiente = lista->primero;
    lista->primero = nuevonodo;
    lista->cantidad++;
    if (lista->cantidad==1) lista->ultimo=nuevonodo;
    return true;
}

bool lista_insertar_ultimo(lista_t *lista, void *dato){
    nodo_lista_t* nuevonodo = malloc(sizeof(nodo_lista_t));
    if (nuevonodo==NULL) return false;
    nuevonodo->datos = dato;
    nuevonodo->siguiente = NULL;
```

```
        if (lista_esta_vacia(lista)){
66      lista->primero = nuevonodo;
        }else{
68      lista->ultimo->siguiente = nuevonodo;
        }
70      lista->ultimo = nuevonodo;
        if (lista->cantidad==0) lista->primero = nuevonodo;
72      lista->cantidad++;
        return true;
74  }

76  void *lista_borrar_primero(lista_t *lista){
        if (lista_esta_vacia(lista)) return NULL;
78      void* retorno = lista_ver_primero(lista);
        nodo_lista_t* provisorio = lista->primero;
80      lista->primero = lista->primero->siguiente;
        lista->cantidad--;
82      if (lista->cantidad==0) lista->ultimo=NULL;
        free(provisorio);
84      return retorno;
    }
86
    void *lista_ver_primero(const lista_t *lista){
88      if (lista->primero==NULL) return NULL;
        return lista->primero->datos;
90  }

92  size_t lista_largo(const lista_t *lista){
        return lista->cantidad;
94  }

96  void lista_destruir(lista_t *lista, void destruir_dato(void *)){
```

```
        while(!lista_esta_vacia(lista)){
98          if (destruir_dato!=NULL) destruir_dato(lista->primero->datos);
            lista_borrar_primero(lista);
100     }
        free(lista);
102 }

104 lista_iter_t *lista_iter_crear(const lista_t *lista){
        lista_iter_t* iter = malloc(sizeof(lista_iter_t));
106     if (iter==NULL) return NULL;
        iter->anterior = NULL;
108     iter->actual = lista->primero;
        return iter;
110 }

112 bool lista_iter_avanzar(lista_iter_t *iter){
        if (iter->actual==NULL) return false;
114     iter->anterior=iter->actual;
        iter->actual=iter->actual->siguiente;
116     return true;
    }
118
    void *lista_iter_ver_actual(const lista_iter_t *iter){
120     if (iter->actual==NULL) return NULL;
        return iter->actual->datos;
122 }

124 bool lista_iter_al_final(const lista_iter_t *iter){
        if (iter->actual==NULL) return true;
126     return false;
    }
128
```

```
    void lista_iter_destruir(lista_iter_t *iter){
130     free(iter);
    }
132
    bool lista_insertar(lista_t *lista, lista_iter_t *iter, void *dato){
134     if (dato==NULL) return false;
        nodo_lista_t* nuevonodo = malloc(sizeof(nodo_lista_t));
136     if (nuevonodo==NULL) return false;
        nuevonodo->datos=dato;
138     if (iter->actual==NULL){
            lista->ultimo=nuevonodo;
140         nuevonodo->siguiente=NULL;
        }else{
142         nuevonodo->siguiente=iter->actual;
        }
144     iter->actual=nuevonodo;
        lista->cantidad++;
146     if (lista->cantidad==1) lista->primero=nuevonodo;
        if (iter->anterior==NULL){
148         lista->primero = nuevonodo;
        }else{
150         iter->anterior->siguiente=nuevonodo;
        }
152     return true;
    }
154
    void *lista_borrar(lista_t *lista, lista_iter_t *iter){
156     if (iter->actual==NULL) return NULL;
        void* dato = iter->actual->datos;
158     if (iter->anterior==NULL){
            lista->primero = iter->actual->siguiente;
160         free(iter->actual);
```

```
            iter->actual = lista->primero;
        }else{
            iter->anterior->siguiente = iter->actual->siguiente;
            free(iter->actual);
            if (iter->anterior->siguiente!=NULL){
                iter->actual = iter->anterior->siguiente;
            }else{iter->actual=NULL;};
        }
        lista->cantidad--;
        if (lista->cantidad==1) lista->ultimo=lista->primero;
        if (lista->cantidad==0) lista->ultimo=NULL;
        return dato;
    }

    void lista_iterar(lista_t *lista, bool (*visitar)(void *dato, void *extra), void
     *extra){
        bool seguir = true;
        void* datoactual;
        lista_iter_t* iter=lista_iter_crear(lista);
        if (iter==NULL) return;
        while(!(lista_iter_al_final(iter)) && seguir){
            datoactual = lista_iter_ver_actual(iter);
            seguir = visitar(datoactual,extra);
            lista_iter_avanzar(iter);
        }
        lista_iter_destruir(iter);
    }

    #endif // LISTA_C
```

```
#define _GNU_SOURCE
#include "comandos.h"
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <stddef.h>
#include <string.h>
#define LMAXCOMANDO 23
#define LPARAMETRO1 11
#define LPARAMETRO2 81
#define LPARAMETRO3 11
#define LPARAMETRO4 4
#define MAXPALABRA 81

typedef struct lectura{
    char* comando;
    char* parametro1;
    char* parametro2;
    char* parametro3;
    char* parametro4;
}lecturaformateada;

void iniciar_parser(lecturaformateada* leer){
    leer->comando = malloc(LMAXCOMANDO*sizeof(char));
    leer->parametro1 = malloc(LPARAMETRO1*sizeof(char));
    leer->parametro2 = malloc(LPARAMETRO2*sizeof(char));
    leer->parametro3 = malloc(LPARAMETRO3*sizeof(char));
    leer->parametro4 = malloc(LPARAMETRO4*sizeof(char));
}

void reiniciar_parser(lecturaformateada* leer){
    leer->comando[0] = '\0';
```

```
        leer->parametro1[0] = '\0';
        leer->parametro2[0] = '\0';
        leer->parametro3[0] = '\0';
        leer->parametro4[0] = '\0';
    }

    void finalizar_parser(lecturaformateada* leer){
        free(leer->comando);
        free(leer->parametro1);
        free(leer->parametro2);
        free(leer->parametro3);
        free(leer->parametro4);
    }

    bool conseguir_comando(lecturaformateada* leer){
        size_t bytesleidos = 0;
        size_t tamlinea = 10;
        int i,j,k;
        if ((leer->comando[0]=fgetc(stdin))==EOF) return false;
        char* linea = malloc(tamlinea*sizeof(char));
        bytesleidos = getline(&linea,&tamlinea,stdin);
        i = 0;
        while(linea[i]!=' ' && linea[i]!='\n'){
            leer->comando[i+1]=linea[i];
            i++;
            if (i>=bytesleidos) break;
        }
        leer->comando[i+1]='\0';
        if (i<bytesleidos){
            k=0;
            while(linea[i]!='\n'){
                i++;
```

```
                j = 0;
66              while(linea[i]!=',' && linea[i]!='\n'){
                    if (k==0) leer->parametro1[j]=linea[i];
68                  if (k==1) leer->parametro2[j]=linea[i];
                    if (k==2) leer->parametro3[j]=linea[i];
70                  if (k==3) leer->parametro4[j]=linea[i];
                    i++;j++;
72                  if (i>=bytesleidos) break;
                }
74              if (k==0) leer->parametro1[j]='\0';
                if (k==1) leer->parametro2[j]='\0';
76              if (k==2) leer->parametro3[j]='\0';
                if (k==3) leer->parametro4[j]='\0';
78              k++;
                if (i>=bytesleidos) break;
            }
80      }
        free(linea);
        return true;
84  }

86  int main(void){
        int devolucion;
88      char* padron;
        base_t* base = inicializar_base_de_datos();
90      sesion_t* sesion = sesion_crear();
        lecturaformateada ingreso;
92      iniciar_parser(&ingreso);
        reiniciar_parser(&ingreso);
94      while (conseguir_comando(&ingreso)){
            //agregar_curso
96          if (strcmp(ingreso.comando,"agregar_curso")==0){
```

```
            devolucion = agregar_curso(ingreso.parametro1,ingreso.parametro2,ing
reso.parametro3,ingreso.parametro4,base);
98          if (devolucion==0) fprintf(stdout,"OK\n");
            if (devolucion==-1) fprintf(stdout,"Error: el curso con id \"%s\" ya existe\n",ing
reso.parametro1);
100         }
            //listar_cursos y listar_cursos_materia
102         if (strcmp(ingreso.comando,"listar_cursos")==0 || strcmp(ingreso.comando,"list
ar_cursos_materia")==0){
            listar_cursos(ingreso.parametro1,base);
104         }
            //inscribir
106         if (strcmp(ingreso.comando,"inscribir")==0){
            devolucion = inscribir(ingreso.parametro1,ingreso.parametro2,base);
108         if (devolucion==1) fprintf(stdout,"OK: padron %s inscripto como regular\n",ing
reso.parametro1);
            if (devolucion==2) fprintf(stdout,"OK: padron %s inscripto como condicional\n",
ingreso.parametro1);
110         if (devolucion==-1) fprintf(stdout,"Error: el padron %s ya esta inscripto en el curso
 \"%s\"\n",ingreso.parametro1,ingreso.parametro2);
            if (devolucion==-2) fprintf(stdout, "Error: el curso con id \"%s\" no existe\n",in
greso.parametro2);
112         }
            //eliminar_curso
114         if (strcmp(ingreso.comando,"eliminar_curso")==0){
            devolucion = eliminar_curso(ingreso.parametro1,base);
116         if (devolucion==0) fprintf(stdout,"OK\n");
            if (devolucion==-1) fprintf(stdout, "Error: el curso con id \"%s\" no existe\n",in
greso.parametro1);
118         }
            //desinscribir
120         if (strcmp(ingreso.comando,"desinscribir")==0){
```

```
            devolucion = desinscribir(ingreso.parametro1,ingreso.parametro2,base
);
122         if (devolucion==0) fprintf(stdout,"OK\n");
            if (devolucion==-1) fprintf(stdout, "Error: el curso con id \"%s\" no existe\n",in
greso.parametro2);
124         if (devolucion==-2) fprintf(stdout, "Error: el padron %s no esta inscripto en el cur
so \"%s\"\n",ingreso.parametro1,ingreso.parametro2);
        }
126     //listar_inscriptos
        if (strcmp(ingreso.comando,"listar_inscriptos")==0){
128         devolucion = listar_inscriptos(ingreso.parametro1,base);
            if (devolucion==-1) fprintf(stdout, "Error: el curso con id \"%s\" no existe\n",in
greso.parametro1);
130     }
        //sesion_iniciar
132     if (strcmp(ingreso.comando,"sesion_iniciar")==0){
            devolucion = sesion_iniciar(sesion,ingreso.parametro1);
134         if (devolucion==0) fprintf(stdout,"OK\n");
            if (devolucion==-1) fprintf(stdout, "Error: ya hay una sesion en curso\n");
136     }
        //sesion_aplicar
138     if (strcmp(ingreso.comando,"sesion_aplicar")==0){
            devolucion = sesion_aplicar(sesion,base);
140         if (devolucion==0) fprintf(stdout,"OK\n");
            if (devolucion==-1) fprintf(stdout,"Error: no hay una sesion en curso\n");
142     }
        //sesion_inscribir
144     if (strcmp(ingreso.comando,"sesion_inscribir")==0){
            devolucion = sesion_inscribir(ingreso.parametro1,base,sesion);
146         padron = obtener_padron(sesion);
            if (devolucion==2) fprintf(stdout,"OK: el padron %s sera inscripto como regular\n"
,padron);
```

```
148                 if (devolucion==1) fprintf(stdout,"OK: el padron %s sera inscripto como condicion
    al\n",padron);
                    if (devolucion==-3) fprintf(stdout,"Error: no hay una sesion en curso\n");
150                 if (devolucion==-2) fprintf(stdout,"Error: el curso con id \"%s\" no existe\n",ing
    reso.parametro1);
                    if (devolucion==-1) fprintf(stdout,"Error: el padron %s ya esta inscripto en el curso
     \"%s\"\n",padron,ingreso.parametro1);
152                 if (devolucion==-4) fprintf(stdout,"Error: la sesion ya contiene una inscripcion al c
    urso\"%s\"\n",ingreso.parametro1);
                    free(padron);
154             }
                //sesion_ver
156             if (strcmp(ingreso.comando,"sesion_ver")==0){
                    sesion_ver(sesion);
158             }
                //sesion_deshacer
160             if (strcmp(ingreso.comando,"sesion_deshacer")==0){
                    devolucion = sesion_deshacer(sesion);
162                 if (devolucion==0) fprintf(stdout,"OK\n");
                    if (devolucion==-1) fprintf(stdout,"Error: no hay una sesion en curso\n");
164                 if (devolucion==-2) fprintf(stdout,"Error: no hay acciones para deshacer\n");
                }
166             reiniciar_parser(&ingreso);
        }
168     finalizar_parser(&ingreso);
        sesion_destruir(sesion);
170     cerrar_base_de_datos(base);
        return 0;
172 }
```

Table of Contents