

Programación en Lenguaje C

Organización de Computadoras

**Universidad Nacional del Sur
Departamento de Ciencias e Ingeniería de la Computación**



Díaz Figueroa, Rodrigo Martín
Figliuolo, Néstor Orlando

Definiciones y especificación de requerimientos

La finalidad del sistema es evaluar expresiones aritméticas que contengan sumas, restas, multiplicaciones y divisiones. Dichas expresiones deberán tener la siguiente forma: (*< operador >* *< operando 1 >* *< operando 2 >* . . . *< operando n >*).

Este sistema esta apuntado a personas que necesitan una solución a la hora de evaluar expresiones del tipo mencionado anteriormente, estos usuarios para poder usar el sistema deben tener conocimientos de como ejecutar un programa por consola y como ingresar los parámetros que sean adecuados.

El sistema cumple con los siguientes requerimientos:

- *Los operadores, paréntesis y operandos están separados por cualquier número de espacios en blanco.*
- *Los operandos son números enteros sin signo, de uno o más dígitos.*
- *Los operadores + y * pueden recibir dos o más operandos.*
- *Las expresiones se interpretan en preorden. Algunos ejemplos de expresiones válidas son:*
- *La expresión (+ (/ 51 37) 6) se interpreta como 51 / 37 + 6.*
- *La expresión (* (+ 9 12 3) 4 5 (- 3 2)) se interpreta como (9 + 12 + 3) * 4 * 5 * (3 - 2)*
- *Para evaluar la expresión aritmética, se debe diseñar e implementar un algoritmo no recursivo, que utilice el TDA pila_t implementado anteriormente. Este algoritmo, además de resolver el problema, debe respetar las siguientes consideraciones:*
- *Si alguno de los operadores no corresponde a un operador válido, debe mostrar un error y finalizar la ejecución con exit status OPRD_INV.*
- *Si la cantidad de operandos es insuficiente para aplicar el operador, debe mostrar un error y abortar con exit status OPND_INSUF.*
- *Si el operador es / o -, y la cantidad de operadores es > 2, debe mostrar un error, y abortar con exit status OPND_DEMAS.*
- *Si alguno de los operandos no corresponde a un número entero válido, debe mostrar un error, y terminar con exit status OPND_INV.*
- *Si al evaluar la expresión no se encuentra un símbolo (, debe mostrar un error y terminar con exit status EXP_MALF.*
- *Si al evaluar la expresión, se encuentra que un (o) no cuenta con su correspondiente) o (, debe mostrar un error y terminar con exit status EXP_MALF.*
- *Si hay un único elemento en la expresión, y es un entero, debe mostrar el resultado y terminar con exit status EXITO. Si el elemento no es un entero, debe terminar con exit status OPND_INV.*
- *La operación aritmética ingresa por la entrada estándar del sistema.*

- Cada operación es evaluada por una función, que toma como parámetro una lista de operandos, contenidos en una estructura de tipo `lista_t`. Por ejemplo, la operación suma (+), es evaluada por la función `int suma(lista_t operandos)`.

El sistema es un desarrollo original y fue desarrollado en parte con el IDE Code::Blocks (www.codeblocks.org/), el editor de texto Komodo Edit (<http://komodoide.com/komodo-edit/>) y compilado con el compilador GCC (<https://gcc.gnu.org/>).

Para poder ejecutar el programa primero se debe descomprimir el archivo comprimido que contiene el código fuente del sistema, cuyo archivo principal es "evaluar.c". Alternativamente se puede clonar el repositorio ubicado en el siguiente enlace: <https://github.com/Makaan/ProyectoC>.

El programa debe ser compilado sobre un sistema operativo GNU/Linux con arquitectura x86 o x86_64 para garantizar su funcionamiento. Para realizar la compilación se debe ejecutar el siguiente comando sobre la carpeta donde se descomprimió el programa:

`"gcc -o evaluar evaluar.c pila.c lista.c"`

Una vez finalizada la compilación el programa puede ser ejecutado con la siguiente sintaxis:

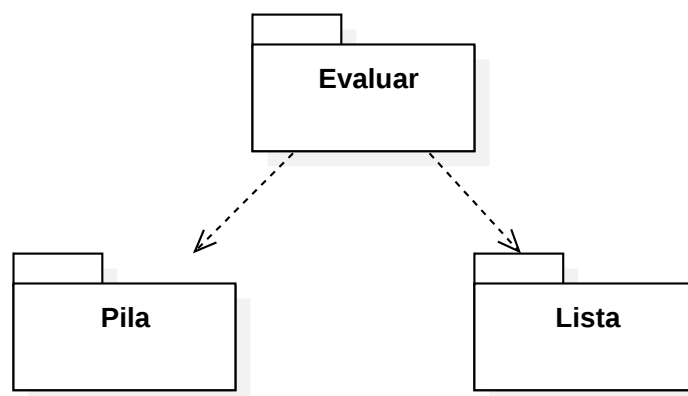
`"/evaluar [-h]"`

Siendo -h un parámetro opcional que solo mostrará por pantalla una ayuda de cómo usar el sistema.

Arquitectura del sistema

El sistema está conformado por tres módulos separados en base a su funcionalidad:

- Evaluar
- Pila
- Lista



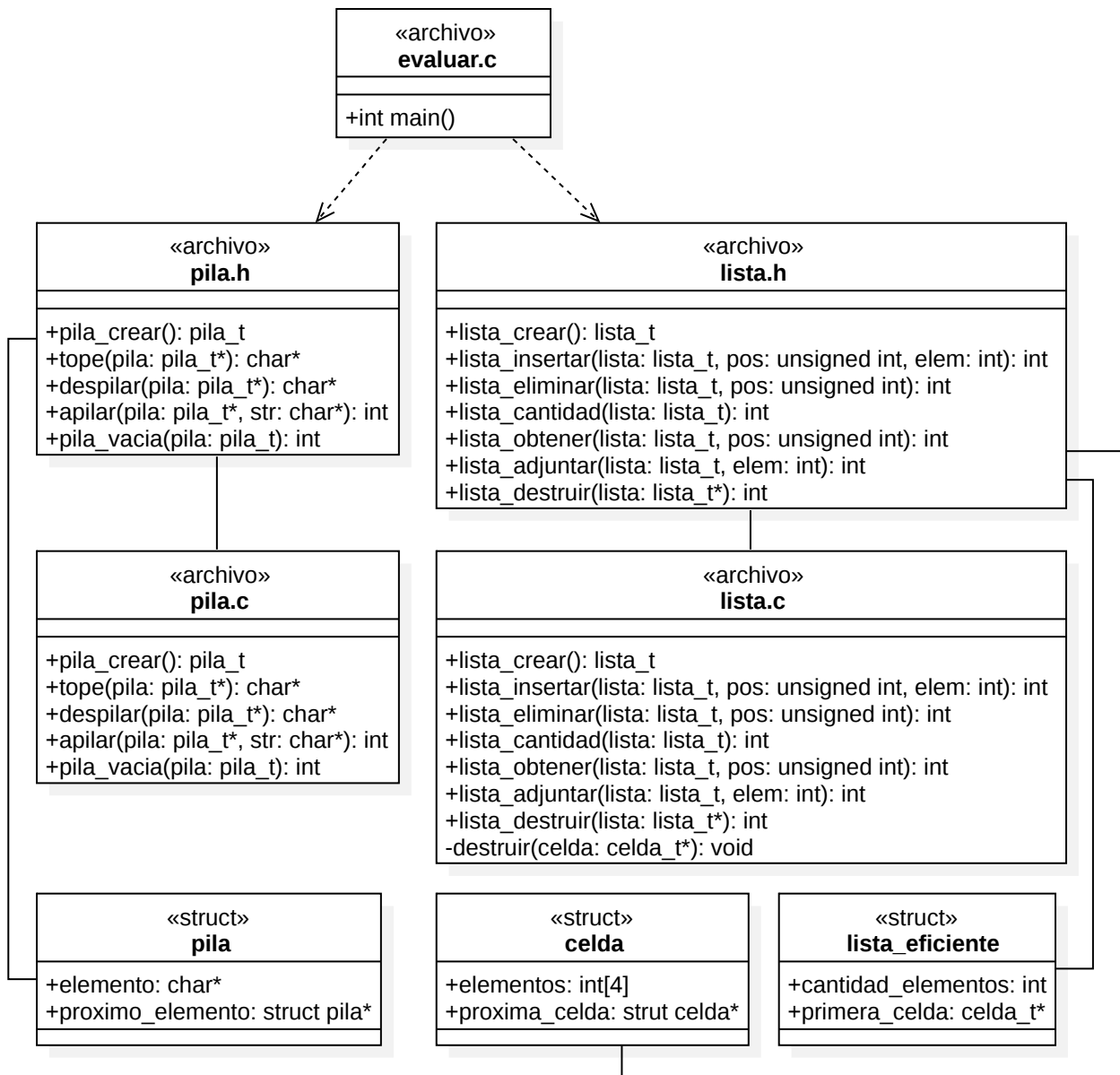
Cada modulo tiene la siguiente descripción:

- Evaluar: aquí esta empaquetado el programa principal, y por ende en el se encuentra toda la implementación concerniente al manejo de los datos de entrada/salida y a la resolución propiamente dicha del problema definido anteriormente. Este modulo depende de los otros dos para funcionar, así también como de la librería de entrada/salida estándar (*stdio*), la librería estándar (*stdlib*) y la librería para manipulación de cadenas de caracteres (*string*), todas provistas por el lenguaje de programación.
- Pila: empaqueta la estructura de datos pila, esta se encarga de modelar una pila con todas sus funcionalidades para que pueda ser utilizada a la hora de resolver el problema almacenando los operadores y paréntesis de la expresión ingresada al sistema. Depende de la librería de entrada/salida estándar (*stdio*) y la librería estándar (*stdlib*).
- Lista: El paquete "Lista" empaqueta todas las funciones necesarias para poder crear y usar estructuras de datos tipo lista y con ellas poder almacenar los números con los cuales se va a evaluar la expresión. Este paquete depende de a librería de entrada/salida estándar (*stdio*) y la librería estándar (*stdlib*).

Todo el sistema esta programado usando el lenguaje de programación C y compilado bajo el sistema operativo GNU/Linux.

Diseño del modelo de datos

El sistema esta diagramado de la siguiente forma:



El sistema usa como dato de entrada la expresión algebraica ingresada por el usuario, internamente se usaran pilas y listas para manejar tanto los operadores como los operandos, y como dato de salida se imprimirá el resultado por pantalla.

Al invocar el programa se le pedirá al usuario que ingrese la expresión algebraica que quiere resolver, si se invoca con el parámetro `-h` se mostrara una ayuda de como usar el sistema. A

continuación cada elemento de la expresión será apilado en una pila, exceptuando los espacios en blanco. Aquí el programa puede reportar las siguientes situaciones de error:

1. Si hay exceso o falta de paréntesis.
2. Si alguno de los operandos/operadores es desconocido.

Luego se aplica un algoritmo que es una variación del Shunting Yard

(https://es.wikipedia.org/wiki/Algoritmo_shunting_yard) y que funciona de la siguiente manera:

- Mientras haya elementos en la expresión
 - Si es un número lo apilo en la pila auxiliar.
 - Si es un paréntesis que cierra lo apilo en la pila auxiliar.
 - Si no es un operador:
 - Si es un paréntesis que abre, y en el tope de la pila auxiliar hay un número:
 - Desapilo de la pila auxiliar.
 - Si en la pila auxiliar hay un paréntesis que cierra:
 - Desapilo el paréntesis que cierra.
 - Apilo el número.
 - Si no es un paréntesis que cierra:
 - Guardo el operador.
 - Mientras no desapilé un paréntesis que cierra de la pila auxiliar:
 - Guardo el número en una lista.
 - Llamo al método que corresponda dependiendo del operador que encontré con la lista de números.
 - Apilo el resultado en la pila auxiliar.
 - Desapilo de la pila.
 - El resultado de la expresión está en el tope de la pila auxiliar.

Especificación técnica:

Evaluar:

int es_digito(char carácter);

Evaluá si el carácter pasado como parámetro es un dígito o no.

Toma como parámetro el carácter a evaluar.

Retorna 1 si es un dígito y 0 en caso contrario.

int carácter_valido(char carácter);

Evaluá si el carácter es un carácter valido, es decir si es "+", "-", "*", "/", "(" o ")".

Toma como parámetro el carácter a evaluar.

Retorna 1 si es un carácter valido y 0 en caso contrario.

int suma(lista_t lista);

Suma todos los elementos en la lista pasada como parámetro.

Toma como parametro la lista con los operando.

Retorna la suma de todos los operandos.

int producto(lista_t lista);

Multiplica todos los elementos en la lista pasada como parámetro.

Toma como parámetro la lista con los operando.

Retorna el producto de todos los operandos.

int resta(lista_t lista);

Resta todos los elementos en la lista pasada como parámetro.

Toma como parámetro la lista con los operando.

Retorna la resta de todos los operandos.

int division(lista_t lista);

Divide todos los elementos en la lista pasada como parámetro.

Toma como parámetro la lista con los operando.

Retorna la división de todos los operandos.

void desapilar_y_evaluar(pila_t pila);

Toma una pila con todos los elementos de la expresión algebraica y calcula el resultado de dicha expresión.

Toma como parámetro la pila con la expresión aritmética.

void apilar_cadena(char* cadena);

Crea una pila con todos los elementos de la expresión algebraica.

Toma como parámetro la cadena de caracteres con la expresión.

void mostrar_ayuda();

Imprime por pantalla la ayuda del programa.

int main(int argc, char argv);**

Método del programa principal.

Pila:

pila_t pila_crear();

Retorna una pila nueva vacía.

char* tope(pila_t pila);

Retorna el string que se encuentra en el tope de la pila. Si la pila se encuentra vacía, aborta su ejecución con exit status PIL_VACIA.
Toma como parámetro una pila a la cual pedirle el tope.
Retorna un puntero a carácter representando el tope de la pila.

char* desapilar(pila_t* pila);

Elimina el string que se encuentra en el tope de la pila y lo retorna. Si la pila se encuentra vacía, aborta su ejecución con exit status PIL_VACIA
Toma como parámetro una pila de la cual se quiere desapilar.
Retorna un puntero a carácter representando lo desapilado.

int apilar(pila_t* pila, char* str);

Inserta el string str en el tope de la pila. Retorna verdadero si la inserción fue exitosa, falso en caso contrario. Si la pila no se encuentra inicializada, finaliza la ejecución con exit status PIL_NO_INI.
Tiene como parámetro un puntero a pila y un string que es lo que se va a apilar.
Retorna 1 si la operación fue completada con éxito.

int pila_vacia(pila_t pila);

Retorna verdadero si la pila esta vacía, falso en caso contrario. Si la pila no se encuentra inicializada, finaliza la ejecución con exit status PIL_NO_INI.
Toma como parametro la pila.
Retorna 1 si la pila esta vacía 0 caso contrario.

Lista:

lista_t lista_crear();

Retorna una nueva lista vacía.

int lista_insertar(lista_t lista, unsigned int pos, int elem);

Inserto un elemento en una posición pasada como parámetro.
Si la posición es mayor a la cantidad de elementos, finaliza la ejecución con error LST_POS_INV.
Si la posición es igual a la cantidad de elementos, inserto al final.
Tiene como parámetro un puntero a una struct lista donde se quiera insertar, un entero sin signo que es la posición donde se quiere insertar y un entero que es el elemento a insertar.
Retorna 1 si la operación se completo con éxito o 0 caso contrario.

int lista_eliminar(lista_t lista, unsigned int pos);

Elimina un elemento de la lista según la posición pasada como parámetro
Si la posición pasada es mayor que la cantidad de elementos, finaliza la ejecución con error LST_POS_INV.
Tiene como parámetro un puntero a struct lista donde se quiera eliminar un elemento y la posición de la lista que se quiere eliminar.
Retorna 1 si la operación se completo exitosamente.

int lista_cantidad(lista_t lista);

Retorna la cantidad de elementos de la lista.

Si la lista no esta inicializada finaliza la ejecución con error LST_NO_INI.

Tiene como parámetro un puntero a un struct lista.

Retorna la cantidad de elementos de la lista.

int lista_obtener(lista_t lista, unsigned int pos);

Retorna el elemento en la posición pasada como parámetro.

Si la posición es mayor a la cantidad de elementos de la lista finaliza la ejecución con error LST_POS_INV .

Tiene como parametro un puntero a un struct lista y un entero que representa la posicion del elemento que se quiere obtener.

Retorna el elemento en la posicion pos.int lista_adjuntar(lista_t lista, int elem);

Agrego un elemento al final de la lista

Si la lista no esta inicializada finaliza la ejecución con error LST_NO_INI.

Tiene como parametro el puntero a la lista y un entero elem que es el elemento a adjuntar.

Retorna 1 si la operación se completo satisfactoriamente.

int lista_destruir(lista_t* lista);

Libero todo el espacio ocupado por la lista, incluyendo las celdas.

Tiene como parametro un puntero a un puntero a lista.

Retorna 1 si la operación se completo satisfactoriamente.

Exit Status	Valor Numerico	Significado
EXITO	EXIT_SUCCESS	Terminación exitosa.
EXP_MALF	2	Expresion malformada,falta o exceso de "()".
LST_NO_INI	3	Lista sin inicializar.
LST_POS_INV	4	Intento de acceso a posición invalida en la lista.
OPND_DEMAS	5	Demasiados operandos para el operador.
OPND_INSUF	6	Insuficientes operandos para el operador.
OPND_INV	7	Operando invalido.
OPRD_INV	8	Operador desconocido o invalido.
PIL_NO_INI	9	Pila no inicializada.
PIL_VACIA	10	Pila vacia.

Conclusiones

Extendimos la funcionalidad del programa original al agregar como correcto las expresiones de tipo (operando), ej. (123) o (((((123)))))) o (+ (123) 123).

El entero mas grande que se puede evaluar

Codigo Fuente:

```
evaluar.c
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <getopt.h>
5  #include "lista.h"
6  #include "pila.h"
7
8  #define CADENA_MAX 32
9
10 const int EXP_MALF=2;
11 const int OPND_DEMAS=5;
12 const int OPND_INSUF=6;
13 const int OPND_INV=7;
14 const int OPRD_INV=8;
15
16 /**
17 Codigo obtenido de: http://stackoverflow.com/a/9660930
18 Funcion que convierte un entero a una cadena de caracteres equivalente
19 **/
20 char* itoa(int i, char b[]){
21     char const digit[] = "0123456789";
22     char* p = b;
23     if(i<0){
24         *p++ = '-';
25         i *= -1;
26     }
27     int shifter = i;
28     do{ //Move to where representation ends
29         ++p;
30         shifter = shifter/10;
31     }while(shifter);
32     *p = '\0';
33     do{ //Move back, inserting digits as u go
34         *--p = digit[i%10];
35         i = i/10;
36     }while(i);
37     return b;
38 }
39 /**
40 Funcion que retorna si el caracter pasado por parametro es un digito de
41 '0' a '9'
42 Retorna 1 si caracter es un digito, 0 en caso contrario.
43 **/
44 int es_digito(char caracter){
45     int toreturn=0;
46     switch(caracter){
47         case '0'...'9': toreturn=1;
48     }
49     return toreturn;
50 }
51 /**
52 Funcion que retorna si el caracter pasado por parametro es un caracter que
53 no
54 es un digito y es valido para una expresion.
55 Retorna 1 si caracter es un caracter, no digito, valido, 0 en caso
56 contrario.
57 **/
```

```

56 int caracter_valido(char caracter){
57     int toreturn=0;
58     switch(caracter){
59         case '+': toreturn=1;
60         case '-': toreturn=1;
61         case '*': toreturn=1;
62         case '/': toreturn=1;
63         case ')': toreturn=1;
64         case '(': toreturn=1;
65     }
66     return toreturn;
67 }
68
69 /**
70 Funcion que suma los elementos de la lista pasada por parametro.
71 Retorna la suma de todos los elementos de la lista lista.
72 Si la cantidad de elementos de la lista es menor a 2, el programa sale con
73 error OPND_INSUF.
74 */
75 int suma(lista_t lista){
76     int resultado=0;
77     int i;
78     int cant=lista_cantidad(lista);
79     //Si la cantidad de elementos de la lista es menor a 2
80     //los operandos son insuficientes y salgo con el error correspondiente
81     if (cant<2) exit(OPND_INSUF);
82     for (i=0;i<cant;i++){
83         resultado=resultado+lista_obtener(lista,i);
84     }
85     //Destruyo la lista luego calcular el resultado
86
87     lista_destruir(&lista);
88
89     return resultado;
90 }
91
92 /**
93 Funcion que multiplica los elementos de la lista pasada por parametro.
94 Retorna el producto de todos los elementos de la lista lista.
95 Si la cantidad de elementos de la lista es menor a 2, el programa sale con
96 error OPND_INSUF.
97 */
98 int producto(lista_t lista){
99     int resultado=1;
100     int i;
101     int cant=lista_cantidad(lista);
102     //Si la cantidad de elementos de la lista es menor a 2
103     //los operandos son insuficientes y salgo con el error correspondiente
104     if (cant<2) exit(OPND_INSUF);
105     for (i=0;i<cant;i++){
106         resultado=resultado*lista_obtener(lista,i);
107     }
108     //Destruyo la lista luego calcular el resultado
109     lista_destruir(&lista);
110     return resultado;
111 }
112
113 /**
114 Funcion que resta los elementos de la lista pasada por parametro.
115 Retorna la resta de los primeros dos elementos de la lista.

```

```

115 Si la cantidad de elementos de la lista es menor a 2, el programa sale con
    error OPND_INSUF.
116 Si la cantidad de elementos de la lista es mayor a 2, el programa sale con
    error OPND_DEMAS.
117 **/
118 int resta(lista_t lista){
119     int resultado=0;
120     int cant=lista_cantidad(lista);
121
122     //Si la cantidad de elementos de la lista es menor a 2
123     //los operandos son insuficientes y salgo con el error correspondiente
124     if (cant<2) exit(OPND_INSUF);
125     //Si la cantidad de elementos de la lista es mayor a 2
126     //los operandos son demasiados y salgo con el error correspondiente
127     if (cant>2) exit(OPND_DEMAS);
128
129     resultado=lista_obtener(lista,0)-lista_obtener(lista,1);
130     //Destruyo la lista luego calcular el resultado
131     lista_destruir(&lista);
132     return resultado;
133 }
134
135 /**
136 Funcion que divide los elementos de la lista pasada por parametro.
137 Retorna la division de los primeros dos elementos de la lista.
138 Si la cantidad de elementos de la lista es menor a 2, el programa sale con
    error OPND_INSUF.
139 Si la cantidad de elementos de la lista es mayor a 2, el programa sale con
    error OPND_DEMAS.
140 **/
141 int division(lista_t lista){
142     int resultado=0;
143     int cant=lista_cantidad(lista);
144
145     //Si la cantidad de elementos de la lista es menor a 2
146     //los operandos son insuficientes y salgo con el error correspondiente
147     if (cant<2) exit(OPND_INSUF);
148     //Si la cantidad de elementos de la lista es mayor a 2
149     //los operandos son demasiados y salgo con el error correspondiente
150     if (cant>2) exit(OPND_DEMAS);
151
152     resultado=lista_obtener(lista,0)/lista_obtener(lista,1);
153     //Destruyo la lista luego calcular el resultado
154     lista_destruir(&lista);
155     return resultado;
156 }
157
158 /**
159 Metodo que toma una pila con los elementos de la expresion ya apilados y
    la evalua utilizando una pila auxiliar.
160 El resultado final se imprime por pantalla
161 **/
162 void desapilar_y_evaluar(pila_t pila){
163
164     char* caracter;
165     char* caracter_aux;
166
167     int resultado=0;
168     //Creo la pila auxiliar
169     pila_t pila_aux=pila_crear();
170

```

```

171     while(!pila_vacia(pila)){
172         caracter=desapilar(&pila);
173         //Si el caracter no es un operador
174         if ( (strcmp(caracter,"+")!=0) && (strcmp(caracter,"*")!=0) &&
175             (strcmp(caracter,"-")!=0) && (strcmp(caracter,"/")!=0)){
176             //Si el caracter es un '(' estoy en el caso de un entero
177             rodeado de parantesis
178             if (strcmp(caracter,"(")==0) {
179                 //Desapilo el entero de la pila auxiliar
180                 caracter_aux=desapilar(&pila_aux);
181                 if (strcmp(tope(pila_aux),")")==0){
182                     //Desapilo el ')' de la pila auxiliar para luego
183                     apilar de nuevo el entero
184                     //y asi eleminar todos los parentesis que rodean al
185                     entero
186                     char* toFree=desapilar(&pila_aux);
187                     free(toFree);
188                     apilar(&pila_aux,caracter_aux);
189                 }
190                 else{
191                     exit(OPRD_INV);
192                 }
193                 free(caracter);
194             }
195             else{
196                 //Si no tengo un entero rodeado de parentesis lo apilo en
197                 la pila auxiliar
198                 apilar(&pila_aux,caracter);
199             }
200         }
201         else{
202             //Creo la lista para luego insertarle los enteros a evaluar
203             lista_t milista=lista_crear();
204             while(strcmp(tope(pila_aux),")")!=0){
205                 //Mientras no haya un ')' en el tope de pila auxiliar
206                 //Sigo desapilando enteros y insertandolos a la lista
207                 caracter_aux=desapilar(&pila_aux);
208                 int num=atoi(caracter_aux);
209                 free(caracter_aux);
210                 lista_adjuntar(milista,num);
211             }
212             //Desapilo el ')' de la pila auxiliar
213             char* toFree=desapilar(&pila_aux);
214             free(toFree);
215             //Dependiendo de el operando llamo a cada funcion con la lista
216             de enteros
217             if (strcmp(caracter,"+")==0) resultado=suma(milista);
218             if (strcmp(caracter,"*")==0) resultado=producto(milista);
219             if (strcmp(caracter,"/")==0) resultado=division(milista);
220             if (strcmp(caracter,"-")==0) resultado=resta(milista);
221             free(caracter);
222             //Convierto el resultado de las funciones de entero a string
223             para luego apilarlo en la pila auxiliar
224             char* resultado_aux=malloc(CADENA_MAX);
225             resultado_aux=itoa(resultado,resultado_aux);
226             apilar(&pila_aux,resultado_aux);
227             //Desapilo el '(' de la pila original
228             toFree=desapilar(&pila);
229             free(toFree);

```

```

225     }
226 }
227 }
228 //Aca el unico elemento de la pila auxiliar es el resultado final
229 //Desapilo el resultado final y lo convierto a entero para mostrarlo
230 char* resuFinal=desapilar(&pila_aux);
231 int toreturn=atoi(resuFinal);
232 free(resuFinal);
233 //Imprimo por pantalla el resultado final
234 printf("Resultado: %d\n",toreturn);
235 }
236
237 /**
238 Metodo que apila solo los elementos de la expresion ingresada por pantalla
239 a una pila.
240 Este metodo no apila espacios en blanco.
241 Si se encuentra con un caracter no valido en la expresion el programa sale
242 con error OPRD_INV.
243 Si uno de los operandos es un numero no entero (es decir que tiene un '.'
244 o una ',' entre sus digitos)
245 el programa sale con error OPND_INV.
246 Si la expresion no esta bien formada el programa sale con error EXP_MALF
247 */
248 void apilar_cadena(char* cadena){
249     //Creo la pila
250     pila_t mipila=pila_crear();
251
252     //Obtengo el largo de la cadena ingresada
253     int largo=strlen(cadena);
254     int i=0;
255     int j;
256     //Utilizo este entero para controlar que cantidad de parentesis sea
257     correcta
258     int cont_parentesis=0;
259
260     while(i<largo-1){
261
262         //Cuento los parentesis
263         if(*(cadena+i)=='(') {
264             cont_parentesis++;
265         }
266         if(*(cadena+i)==')') {
267             cont_parentesis--;
268         }
269
270         //Si no es espacio...
271         if (*(cadena+i)!=' '){
272
273             //Si es un digito...
274             if ( es_digito(*(cadena+i)) ){
275                 //Reservo memoria para la nueva cadena que es un número
276                 char* num=malloc(CADENA_MAX);
277                 j=0;
278                 //Mientras sea digito lo guardo en una string aux
279                 while(es_digito(*(cadena+i))){
280                     *(num+j)=*(cadena+i);
281                     j++;
282                     i++;
283                 }

```



```

281          //Si se introduce un numero no entreo (con coma o punto)
sale con error
282          if ( (*(cadena+i)=='.')||(*(cadena+i)==',' ) )
283              //Si el operando no es un entero salgo con el error
correspondiente
284              exit (OPND_INV);
285              //Apilo el numero
286              apilar(&mipila,num);
287          }
288          else{
289              //Apilo el caracter valido que no es numero
290              if (caracter_valido(*(cadena+i))) {
291                  char* paraApilar=malloc(sizeof(char));
292                  *paraApilar=(*(cadena+i));
293                  apilar(&mipila,paraApilar);
294                  i++;
295              }
296              else
297                  //Si el caracter no es valido salgo con el error
correspondiente
298                  exit (OPRD_INV);
299              }
300          }
301      }
302      else{
303          //Aumento i cuando el caracter es un espacio
304          i++;
305      }
306  }
307  //Si la cantidad de parentesis no es la correcta salgo con el error
correspondiente
308  if(cont_parentesis!=0) exit(EXP_MALF);
309
310  //Llamo al metodo desapilar_y_evaluar para evaluar la expresion
311  desapilar_y_evaluar(mipila);
312  }
313
314  void mostrar_ayuda(){
315      printf("El programa evalúa expresiones aritméticas por entrada
estándar interpretadas en preorden.\n");
316      printf("Las expresiones aritméticas son suma, resta, división y
multiplicación.\n\n");
317      printf("Las expresiones aritméticas utilizan la siguiente
sintaxis:\n");
318      printf("<operador> <operando1> <operando2> ... <operandoN>)\n");
319      printf("Donde un <operandoI> puede ser de la forma:\n");
320      printf("<operando>)\n");
321      printf("Con <operando> un número entero, que va a ser procesado como
<operando> sin importar\n");
322      printf("la cantidad de parentesis que lo rodea, siempre y cuando los
parentesis esten bien formados.\n\n");
323      printf("Todas las operaciones deben recibir al menos dos operandos
para poder ser evaluadas.\n");
324      printf("Las operaciones suma y multiplicación pueden recibir más de
dos operandos.\n\n");
325      printf("Un caso especial de expresion a evaluar es:\n");
326      printf("<operando>)\n");
327      printf("Donde <operando> es un número entero, y es el mismo resultado
de la expresión.\n\n");

```

```

329     printf("El parámetro -h muestra esta ayuda.\n");
330 }
331
332 int main(int argc, char** argv){
333     int argumento;
334     while ((argumento = getopt (argc, argv, "h")) != -1){
335         switch (argumento){
336             case 'h':{
337                 mostrar_ayuda();
338                 exit(0);
339             }
340             default: {
341                 printf("Argumentos incorrectos\n");
342                 mostrar_ayuda();
343                 exit(0);
344             }
345         }
346     }
347     char *cadena = malloc (512);
348     printf("Ingrese la expresión\n");
349     fgets (cadena, 512, stdin);
350     apilar_cadena(cadena);
351     free(cadena);
352     printf("\nFin del programa\n");
353     return 0;
354 }

```

pila.h

```

1  #ifndef PILA_H_INCLUDED
2  #define PILA_H_INCLUDED
3
4  typedef struct pila {
5      char* elemento;
6      struct pila* proximo_elemento;
7  } *pila_t;
8
9  /**Retorna una pila nueva vacía**/
10 pila_t pila_crear();
11
12 /**Retorna el string que se encuentra en el tope de la pila. Si la
13 pila se encuentra vacía, aborta su ejecucion con exit status PIL_VACIA**/
14 char* tope(pila_t pila);
15
16 /**Elimina el string que se encuentra en el tope de la pila y lo retorna.
17 Si la
18 pila se encuentra vacía, aborta su ejecucion con exit status PIL_VACIA**/
19 char* desapilar(pila_t* pila);
20
21 /**Inserta el string str en el tope de la pila. Retorna verdadero si la
22 insercion fue exitosa,
23 falso en caso contrario. Si la pila no se encuentra inicializada, finaliza
24 la ejecucion con
25 exit status PIL_NO_INI**/
26 int apilar(pila_t* pila, char* str);
27
28 /**Retorna verdadero si la pila esta vacía, falso (0) en caso contrario.
29 Si la pila
30 no se encuentra inicializada, finaliza la ejecución con exit status
31 PIL_NO_INI**/
32 int pila_vacia(pila_t pila);

```

```

28
29 #endif // PILA_H_INCLUDED

```

lista.h

```

1  #ifndef LISTA_H_INCLUDED
2  #define LISTA_H_INCLUDED
3
4  //Estructura celda que modela un componente de la lista que almacena hasta
   cuatro elementos.
5  typedef struct celda {
6      int elementos[4];
7      struct celda* proxima_celda;
8  } celda_t;
9
10 //Cabecera de la estructura lista, posee una referencia a la primera celda
   y la cantidad de elementos de la lista.
11 typedef struct lista_eficiente {
12     unsigned int cantidad_elementos;
13     celda_t* primera_celda;
14 } *lista_t;
15
16 //Retorna una nueva lista vacia
17 lista_t lista_crear();
18
19 //Inserto un elemento en una posición pasada como parametro
20 //Si la posicion es mayor a la cantidad de elementos, finaliza la
   ejecucion con error LST_POS_INV
21 //Si la posicion es igual a la cantidad de elementos, inserto al final.
22 int lista_insertar(lista_t lista, unsigned int pos, int elem);
23
24 //Elimina un elemento de la lista segun la posicion pasada como parametro
25 //Si la posicion pasada es mayor que la cantidad de elementos, finaliza la
   ejecucion con error LST_POS_INV
26 int lista_eliminar(lista_t lista, unsigned int pos);
27
28 //Retorna la cantidad de elementos de la lista.
29 //Si la lista no esta inicializada finaliza la ejecucion con error
   LST_NO_INI
30 int lista_cantidad(lista_t lista);
31
32 //Retorna el elemento en la posicion pasada como parametro
33 //Si la posicion es mayor a la cantidad de elementos de la lista finaliza
   la ejecucion con error LST_POS_INV
34 int lista_obtener(lista_t lista, unsigned int pos);
35
36 //Agrego un elemento al final de la lista
37 //Si la lista no esta inicializada finaliza la ejecucion con error
   LST_NO_INI
38 int lista_adjuntar(lista_t lista, int elem);
39
40 //Libero todo el espacio ocupado por la lista, incluyendo las celdas.
41 int lista_destruir(lista_t* lista);
42
43 #endif // LISTA_H_INCLUDED

```

pila.c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include "pila.h"

```

```

4
5     const int PIL_NO_INI = 9;
6     const int PIL_VACIA = 10;
7
8     /**Retorna una pila nueva vacía**/
9     pila_t pila_crear(){
10         return NULL;
11     }
12
13     /**Retorna el string que se encuentra en el tope de la pila. Si la
14     pila se encuentra vacia, aborta su ejecucion con exit status PIL_VACIA**/
15     char* tope(pila_t pila){
16         if (pila_vacia(pila))
17             exit(PIL_VACIA);
18         else
19             return pila->elemento;
20     }
21
22     /**Elimina el string que se encuentra en el tope de la pila y lo retorna.
23     Si la
24     pila se encuentra vacia, aborta su ejecucion con exit status PIL_VACIA**/
25     char* desapilar(pila_t* pila){
26         char* aux;
27         pila_t tofree;
28         if (pila_vacia(*pila)){
29             exit(PIL_VACIA);
30         }
31         tofree=(*pila);
32         aux=(*pila)->elemento;
33
34         if (((*pila)->proximo_elemento)!=NULL)
35             (*pila)=((*pila)->proximo_elemento);
36         else
37             (*pila)=NULL;
38
39         free(tofree);
40         return aux;
41     }
42
43     /**Inserta el string str en el tope de la pila. Retorna verdadero si la
44     insercion fue exitosa,
45     falso en caso contrario. Si la pila no se encuentra inicializada, finaliza
46     la ejecucion con
47     exit status PIL_NO_INI**/
48     int apilar(pila_t* pila, char* str){
49         if(pila==NULL) {
50             exit(PIL_NO_INI);
51         }
52         int toreturn=1;
53         if (str!=NULL){
54             pila_t nueva=(pila_t)malloc(sizeof(pila_t));
55             nueva->elemento=str;
56             nueva->proximo_elemento=(*pila);
57             (*pila)=nueva;
58             toreturn=0;
59         }
60         return toreturn;
61     }
62
63     /**Retorna verdadero si la pila esta vacia, falso en caso contrario. Si la
64     pila

```

```

61     no se encuentra inicializada, finaliza la ejecucion con exit status
    PIL_NO_INI**/
62     int pila_vacia(pila_t pila){
63         if (pila==NULL)
64             return 0;
65         else
66             return 1;
67     }

```

lista.c

```

1     #include <stdio.h>
2     #include <stdlib.h>
3     #include "lista.h"
4
5     const int LST_NO_INI=3;
6     const int LST_POS_INV=4;
7
8
9
10    //Retorna una nueva lista vacia.
11    lista_t lista_crear() {
12        //Asigno la cantidad de memoria necesaria.
13        lista_t lista=(lista_t) malloc(sizeof(struct lista_eficiente));
14
15        //Creo la lista.
16        lista->cantidad_elementos=0;
17        lista->primera_celda=NULL;
18
19        return lista;
20    }
21
22    //Inserto un elemento en una posicion pasada como parametro
23    //Si la posicion es mayor a la cantidad de elementos, finaliza la
    ejecucion con error LST_POS_INV
24    //Si la posicion es igual a la cantidad de elementos, inserto al final.
25    int lista_insertar(lista_t lista, unsigned int pos, int elem) {
26        //Si la posicion es mayor que la cantidad de elementos salgo con
    error de posicion invalida.
27        if(pos>lista->cantidad_elementos)
28            exit(LST_POS_INV);
29
30        //Obtengo la celda donde debo insertar.
31        int posCelda=pos/4;
32        if(lista->primera_celda==NULL) {
33            lista->primera_celda=(celda_t*) malloc(sizeof(celda_t));
34            lista->primera_celda->proxima_celda=NULL;
35        }
36        celda_t* celda_actual=lista->primera_celda;
37        //Recorro hasta encontrar la celda donde agregar.
38        int i;
39        for(i=0;i<posCelda;i++) {
40            //Si esa celda no existe la creo.
41            if((celda_actual->proxima_celda)==NULL) {
42                celda_t* nuevaCelda=(celda_t*)malloc(sizeof(celda_t));
43                nuevaCelda->proxima_celda=NULL;
44                celda_actual->proxima_celda=nuevaCelda;
45            }
46            celda_actual=celda_actual->proxima_celda;
47        }
48    }

```

```

49
50     int posArreglo=pos%4;
51     if(pos==(lista->cantidad_elementos))
52         lista->cantidad_elementos++;
53     celda_actual->elementos[posArreglo]=elem;
54
55
56     return 0;
57 }
58
59 //Elimina un elemento de la lista segun la posicion pasada como parametro
60 //Si la posicion pasada es mayor que la cantidad de elementos, finaliza la
ejecucion con error LST_POS_INV
61 int lista_eliminar(lista_t lista, unsigned int pos) {
62     //Salgo con error si la posicion no existe.
63     if(pos>=lista->cantidad_elementos) {
64         exit(LST_POS_INV);
65     }
66     //Obtengo la celda donde voy a eliminar
67     int posCelda=pos/4;
68     celda_t* celda_actual=lista->primera_celda;
69     int i;
70     //Recorro hasta llegar a esa celda
71     for(i=0;i<posCelda;i++) {
72         celda_actual=celda_actual->proxima_celda;
73     }
74     //Obtengo la posicion del arreglo de esa celda donde eliminar.
75     int posArreglo=pos%4;
76     //Acomodo todos los elementos restantes del arreglo para cerrar el
espacio creado por elemento eliminado.
77     while(pos<lista->cantidad_elementos) {
78         //Muevo cada elemento i+1 al i en el arreglo de la celda.
79         for(;posArreglo<3 && (pos<lista-
>cantidad_elementos);posArreglo++) {
80             celda_actual->elementos[posArreglo]=celda_actual-
>elementos[posArreglo+1];
81             pos++;
82         }
83         //Cuando termino con el arreglo de esa celda me muevo a la
celda siguiente1
84         if(pos<lista->cantidad_elementos) {
85             celda_actual->elementos[3]=celda_actual->proxima_celda-
>elementos[0];
86             celda_actual=celda_actual->proxima_celda;
87             posArreglo=0;
88             pos++;
89         }
90     }
91     lista->cantidad_elementos--;
92     return 0;
93 }
94
95
96 //Retorna la cantidad de elementos de la lista.
97 //Si la lista no esta inicializada finaliza la ejecucion con error
LST_NO_INI
98 int lista_cantidad(lista_t lista) {
99     //Si la lista no esta inicializada corta la ejecucion y sale con
error.
100     if(lista==NULL) {
101         exit(LST_NO_INI);

```

```

102     }
103     return lista->cantidad_elementos;
104 }
105
106 //Retorna el elemento en la posicion pasada como parametro
107 //Si la posicion es mayor a la cantidad de elementos de la lista finaliza
108 //la ejecucion con error LST_POS_INV
109 int lista_obtener(lista_t lista, unsigned int pos) {
110     //Si la lista no esta inicializada salgo con error.
111     if(lista==NULL){
112         exit(LST_NO_INI);
113     }
114     //Si la posicion no es valida corta la ejecucion con error.
115     if(pos>(lista->cantidad_elementos-1)) {
116         exit(LST_POS_INV);
117     }
118     celda_t* celda_actual=lista->primera_celda;
119     //Recorro la lista tantas veces como indique el parametro "pos".
120     int i;
121     for(i=0;i<pos/4;i++) {
122         celda_actual=celda_actual->proxima_celda;
123     }
124     return celda_actual->elementos[pos%4];
125 }
126
127 int lista_adjuntar(lista_t lista, int elem) {
128     //Uso el metodo lista_insertar con la cantidad de elementos de la
129     //lista como posicion
130     int to_return=lista_insertar(lista,(lista-
131     >cantidad_elementos),elem);
132     return to_return;
133 }
134
135 //Metodo recursivo que recorre todas las celdas y les hace free cuando
136 //vuelve de la recursion
137 void destruir(celda_t* celda) {
138     //Si hay mas celdas llamo recursivamente
139     if((celda->proxima_celda)!=NULL){
140         destruir(celda->proxima_celda);
141     }
142     celda->proxima_celda=NULL;
143     free(celda);
144     celda=NULL;
145 }
146
147 //Agrego un elemento al final de la lista
148 //Si la lista no esta inicializada finaliza la ejecucion con error
149 //LST_NO_INI
150 int lista_destruir(lista_t* lista) {
151     if((*lista)->primera_celda==NULL) {
152         exit(LST_NO_INI);
153     }
154     //Obtengo la primera celda.
155     celda_t* celda=(*lista)->primera_celda;
156     //Lamo recursivamente para liberar el espacio de las celdas.
157     destruir(celda);
158     free(*lista);
159     *lista=NULL;
160     lista=NULL;
161     return 0;
162 }

```