11 de Octubre del 2016

Programación en Lenguaje C

Organización de Computadoras

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

Diaz Figueroa, Rodrigo Martin
Figliuolo, Nestor 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 cómo 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 origina 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 ejectuar el programa primero se debe descomprimir el archivo comprimido que contiene el codigo 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 compilacion se debe ejecutar el siguiente comando sobre la carpeta donde se descomprimio el programa:
“gcc -o evaluar evaluar.c pila.c lista.c”

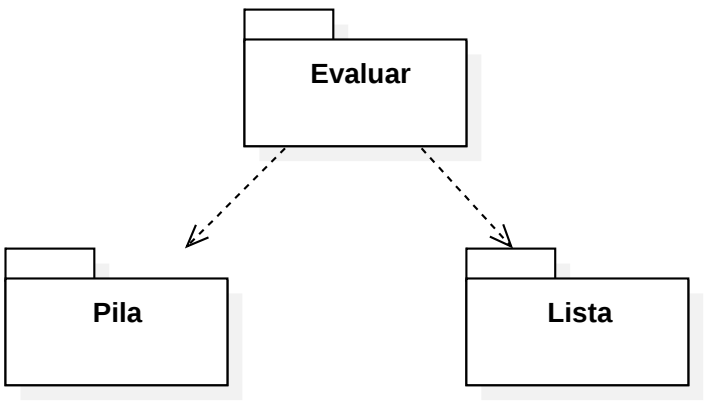
Una vez finalizada la compilacion el programa puede ser ejecutado con la siguiente sintaxis:
“./evaluar [-h]”

Siendo -h un parametro opcional que solo mostrara por pantalla una ayuda de como usar el sistema.

Arquitectura del sistema

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

- Evaluar
- Pila
- Lista



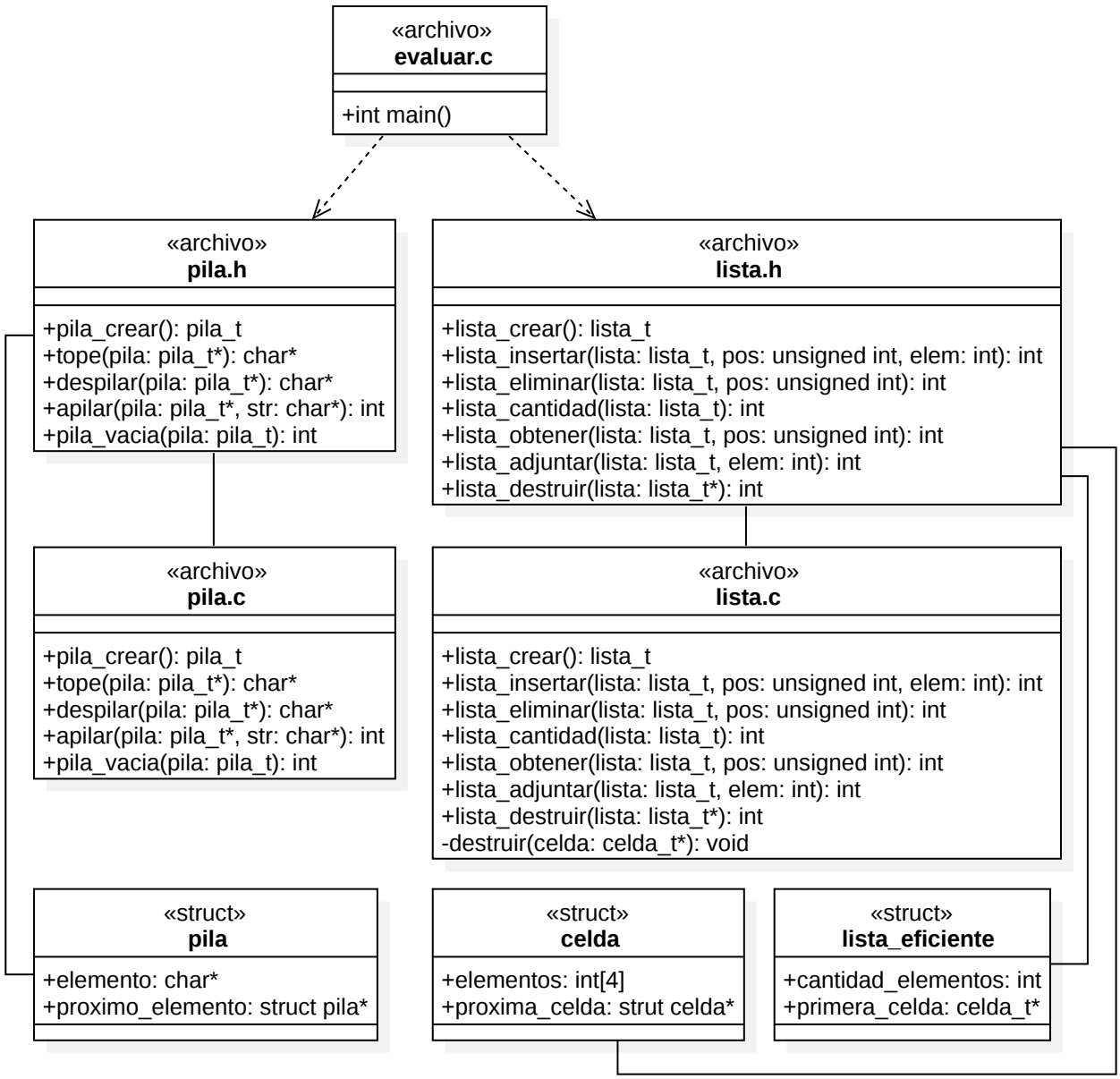
Cada modulo tiene la siguiente descripción:

- 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 sera 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 numero 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 numero:
      - Desapilo de la pila auxiliar.
      - Si en la pila auxiliar hay un paréntesis que cierra:
        - Desapilo el paréntesis que cierra.
        - Apilo el numero.
      - Si no es un paréntesis que cierra:
        - Guardo el operador.
        - Mientras no desapilé un paréntesis que cierra de la pila auxiliar:
          - Guardo el numero 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 esta 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 vaciá, 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 vacia, 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 vacia, 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 vacia 0 caso contrario.

Lista:

`lista_t lista_crear();`
Retorna una nueva lista vaciá.

`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 ejecucion 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 ejecucion 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
      '0' a '9'
41    Retorna 1 si caracter es un digito, 0 en caso contrario.
42    **/
43    int es_digito(char caracter){
44        int toreturn=0;
45        switch(caracter){
46            case '0'...'9': toreturn=1;
47        }
48        return toreturn;
49    }
50
51    /**
52    Funcion que retorna si el caracter pasado por parametro es un caracter que
      no
53    es un digito y es valido para una expresion.
54    Retorna 1 si caracter es un caracter, no digito, valido, 0 en caso
      contrario.
55    **/
```

```
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
    error OPND_INSUF.
73  **/
74  int suma(lista_t lista){
75      int resultado=0;
76      int i;
77      int cant=lista_cantidad(lista);
78      //Si la cantidad de elementos de la lista es menor a 2
79      //los operandos son insuficientes y salgo con el error correspondiente
80      if (cant<2) exit(OPND_INSUF);
81
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
    error OPND_INSUF.
96  **/
97  int producto(lista_t lista){
98      int resultado=1;
99      int i;
100     int cant=lista_cantidad(lista);
101     //Si la cantidad de elementos de la lista es menor a 2
102     //los operandos son insuficientes y salgo con el error correspondiente
103     if (cant<2) exit(OPND_INSUF);
104     for (i=0;i<cant;i++){
105         resultado=resultado*lista_obtener(lista,i);
106     }
107     //Destruyo la lista luego calcular el resultado
108     lista_destruir(&lista);
109     return resultado;
110 }
111
112 /**
113 Funcion que resta los elementos de la lista pasada por parametro.
114 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) &&
      (strcmp(caracter,"-")!=0) && (strcmp(caracter,"/")!=0)){
175               //Si el caracter es un '(' estoy en el caso de un entero
      rodeado de parantesis
176               if (strcmp(caracter,"(")==0) {
177                   //Desapilo el entero de la pila auxiliar
178                   caracter_aux=desapilar(&pila_aux);
179                   if (strcmp(tope(pila_aux),")")==0){
180                       //Desapilo el ')' de la pila auxiliar para luego
      apilar de nuevo el entero
181                       //y asi eleminar todos los parentesis que rodean al
      entero
182                       char* toFree=desapilar(&pila_aux);
183                       free(toFree);
184                       apilar(&pila_aux,caracter_aux);
185                   }
186                   else{
187                       exit(OPRD_INV);
188                   }
189                   free(caracter);
190               }
191               else{
192                   //Si no tengo un entero rodeado de parentesis lo apilo en
      la pila auxiliar
193                   apilar(&pila_aux,caracter);
194               }
195           }
196           else{
197               //Creo la lista para luego insertarle los enteros a evaluar
198
199               lista_t milista=lista_crear();
200               while(strcmp(tope(pila_aux),")")!=0){
201                   //Mintras no haya un ')' en el tope de pila auxiliar
202                   //Sigo desapilando enteros y insertandolos a la lista
203                   caracter_aux=desapilar(&pila_aux);
204                   int num=atoi(caracter_aux);
205                   free(caracter_aux);
206                   lista_adjuntar(milista,num);
207               }
208               //Desapilo el ')' de la pila auxiliar
209               char* toFree=desapilar(&pila_aux);
210               free(toFree);
211               //Dependiendo de el operando llamo a cada funcion con la lista
      de enteros
212               if (strcmp(caracter,"+")==0) resultado=suma(milista);
213               if (strcmp(caracter,"*")==0) resultado=producto(milista);
214               if (strcmp(caracter,"/")==0) resultado=division(milista);
215               if (strcmp(caracter,"-")==0) resultado=resta(milista);
216               free(caracter);
217               //Convierto el resultado de las funciones de entero a string
      para luego apilarlo en la pila auxiliar
218               char* resultado_aux=malloc(CADENA_MAX);
219               resultado_aux=itoa(resultado,resultado_aux);
220               apilar(&pila_aux,resultado_aux);
221               //Desapilo el '(' de la pila original
222               toFree=desapilar(&pila);
223               free(toFree);
224
```

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

```
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     }
308     //Si la cantidad de parentesis no es la correcta salgo con el error
    correspondiente
309     if(cont_parentesis!=0) exit(EXP_MALF);
310
311     //Llamo al metodo desapilar_y_evaluar para evaluar la expresion
    apilada
312     desapilar_y_evaluar(mipila);
313 }
314
315 void mostrar_ayuda(){
316     printf("El programa evalúa expresiones aritméticas por entrada
    estándar interpretadas en preorden.\n");
317     printf("Las expresiones aritméticas son suma, resta, división y
    multiplicación.\n\n");
318     printf("Las expresiones aritméticas utilizan la siguiente
    sintaxis:\n");
319     printf("(<operador> <operando1> <operando2> ... <operandoN>)\n");
320     printf("Donde un <operandoI> puede ser de la forma:\n");
321     printf("(<operando>)\n");
322     printf("Con <operando> un número entero, que va a ser procesado como
    <operando> sin importar\n");
323     printf("la cantidad de parentesis que lo rodea, siempre y cuando los
    parentesis esten bien formados.\n\n");
324     printf("Todas las operaciones deben recibir al menos dos operandos
    para poder ser evaluadas.\n");
325     printf("Las operaciones suma y multiplicación pueden recibir más de
    dos operandos.\n\n");
326     printf("Un caso especial de expresion a evaluar es:\n");
327     printf("(<operando>)\n");
328     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 vacia, 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.
        Si la
17      pila se encuentra vacia, aborta su ejecucion con exit status PIL_VACIA**/
18      char* desapilar(pila_t* pila);
19
20      /**Inserta el string str en el tope de la pila. Retorna verdadero si la
        insercion fue exitosa,
21      falso en caso contrario. Si la pila no se encuentra inicializada, finaliza
        la ejecucion con
22      exit status PIL_NO_INI**/
23      int apilar(pila_t* pila, char* str);
24
25      /**Retorna verdadero si la pila esta vacia, falso (0) en caso contrario.
        Si la pila
26      no se encuentra inicializada, finaliza la ejecución con exit status
        PIL_NO_INI**/
27      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 enceuntra 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.
      Si la
23    pila se encuentra vacia, aborta su ejecucion con exit status PIL_VACIA**/
24    char* desapilar(pila_t* pila){
25       char* aux;
26       pila_t tofree;
27       if (pila_vacia(*pila)){
28            exit(PIL_VACIA);
29       }
30       tofree=(*pila);
31       aux=(*pila)->elemento;
32
33       if (((*pila)->proximo_elemento)!=NULL)
34           (*pila)=((*pila)->proximo_elemento);
35       else
36           (*pila)=NULL;
37
38       free(tofree);
39       return aux;
40    }
41
42    /**Inserta el string str en el tope de la pila. Retorna verdadero si la
      insercion fue exitosa,
43    falso en caso contrario. Si la pila no se encuentra inicializada, finaliza
      la ejecucion con
44    exit status PIL_NO_INI**/
45    int apilar(pila_t* pila, char* str){
46       if(pila==NULL) {
47          exit(PIL_NO_INI);
48       }
49       int toreturn=1;
50       if (str!=NULL){
51          pila_t nueva=(pila_t)malloc(sizeof(pila_t));
52          nueva->elemento=str;
53          nueva->proximo_elemento=(*pila);
54          (*pila)=nueva;
55          toreturn=0;
56        }
57       return toreturn;
58    }
59
60    /**Retorna verdadero si la pila esta vacia, falso en caso contrario. Si la
      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                }
47                celda_actual=celda_actual->proxima_celda;
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
      la ejecucion con error LST_POS_INV
108   int lista_obtener(lista_t lista, unsigned int pos) {
109         //Si la lista no esta inicializada salgo con error.
110         if(lista==NULL){
111               exit(LST_NO_INI);
112         }
113         //Si la posicion no es valida corta la ejecucion con error.
114         if(pos>(lista->cantidad_elementos-1)) {
115               exit(LST_POS_INV);
116         }
117         celda_t* celda_actual=lista->primera_celda;
118         //Recorro la lista tantas veces como indique el parametro "pos".
119         int i;
120         for(i=0;i<pos/4;i++) {
121               celda_actual=celda_actual->proxima_celda;
122         }
123         return celda_actual->elementos[pos%4];
124   }
125
126   int lista_adjuntar(lista_t lista, int elem) {
127         //Uso el metodo lista_insertar con la cantidad de elementos de la
      lista como posicion
128         int to_return=lista_insertar(lista,(lista-
      >cantidad_elementos),elem);
129         return to_return;
130   }
131
132   //Metodo recursivo que recorre todas las celdas y les hace free cuando
      vuelve de la recursion
133   void destruir(celda_t* celda) {
134         //Si hay mas celdas llamo recursivamente
135         if((celda->proxima_celda)!=NULL){
136               destruir(celda->proxima_celda);
137       }
138       celda->proxima_celda=NULL;
139         free(celda);
140         celda=NULL;
141   }
142
143   //Agrego un elemento al final de la lista
144   //Si la lista no esta inicializada finaliza la ejecucion con error
      LST_NO_INI
145   int lista_destruir(lista_t* lista) {
146         if((*lista)->primera_celda==NULL) {
147               exit(LST_NO_INI);
148         }
149         //Obtengo la primera celda.
150         celda_t* celda=(*lista)->primera_celda;
151         //Lamo recursivamente para liberar el espacio de las celdas.
152         destruir(celda);
153         free(*lista);
154         *lista=NULL;
155         lista=NULL;
156         return 0;
157   }
```