# Organización de Computadoras

Departamento de Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur
**Segundo Cuatrimestre de 2016**


Proyecto N$^{o}$ 1
**Programación en Lenguaje C**

El objetivo principal de este proyecto es implementar en lenguaje C, un programa que evalúe expresiones aritméticas expresadas en preorden.

Con este objetivo debe implementarse:

- un TDA Lista, utilizando memoria dinámica mediante las funciones de libería `void free(*void ptr)` y `void *malloc(size_t size)`.
- un TDA Pila para almacenar *strings* de caracteres.
- un programa que evalúe expresiones aritméticas por entrada estándar, utilizando los TDA anteriormente implementados.

# 1. TDA Lista

Implementar un TDA Lista en lenguaje C, cuyos elementos sean arreglos de enteros. La lista debe ser simplemente enlazada, y su implementación realizada con punteros. El TDA contará con las siguientes operaciones:

- `lista_t lista_crear()` Crea una lista vacía y la retorna.
- `int lista_insertar(lista_t lista, unsigned int pos, int elem)` Inserta el elemento `elem` en la posición `pos` de la lista. Sobrescribe el valor existente en la posición indicada, o agrega un nuevo elemento si la posición coincide con la cantidad de elementos. Si procede exitosamente retorna verdadero. En otro caso retorna falso.
- `int lista_eliminar(lista_t lista, unsigned int pos)` Elimina el elemento en la posición `pos`. Reacomoda la lista adecuadamente al eliminar en posiciones intermedias. Retorna verdadero si procede con éxito, falso en caso contrario. Si la posición no es válida, aborta con *exit status* `LST_POS_INV`.
- `int lista_cantidad(lista_t lista)` Retorna la cantidad de elementos de la lista. Si la lista no está inicializada, el programa aborta con *exit status* `LST_NO_INI`.
- `int lista_obtener(lista_t lista, unsigned int pos)` Retorna el elemento en la posición `pos` de la lista. Si la posición no es válida, aborta con *exit status* `LST_POS_INV`.
- `int lista_adjuntar(lista_t lista, int elem)` Inserta el elemento `elem` en la última posición de la lista. Si la lista no está inicializada, aborta con *exit status* `LST_NO_INI`.
- `int lista_destruir(lista_t* lista)` Libera la memoria ocupada por la lista y le asigna `NULL`. Retorna verdadero en caso de éxito. Si la lista no está inicializada, aborta con *exit status* `LST_NO_INI`.

Donde el tipo `lista_t` está definido de la siguiente manera:

```
typedef struct lista_eficiente {
    unsigned int cantidad_elementos;
    celda_t* primera_celda;
} *lista_t;

typedef struct celda {
    int elementos[4];
    struct celda* proxima_celda;
} celda_t;
```

# 2. TDA Pila

Implementar un TDA Pila en lenguaje C, cuyos elementos sean *strings* (`char*`). La pila debe implementarse como una lista dinámica, simplemente enlazada, sin centinela. La implementación debe proveer las operaciones:

1. `pila_t pila_crear()` Retorna una pila nueva vacía.
2. `char* tope(pila_t pila)` Retorna el string que se encuentra en el tope de la pila. Si la pila se encuentre vacía, aborta su ejecución con *exit status* `PIL_VACIA`.
3. `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`.
4. `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`.
5. `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`.

Donde el tipo `pila_t` está definido de la siguiente manera:

```
typedef struct pila {
    char* elemento;
    struct pila* proximo_elemento;
} *pila_t;
```

# 3. Programa principal

Implementar un evaluador de las expresiones aritméticas suma, resta, división y multiplicación ($+$, $-$, $/$, $*$), utilizando los tipos de datos pila y lista implementados anteriormente. Las expresiones aritméticas utilizan la siguiente sintaxis:

$$(< operador > < operando_1 > < operando_2 > \ldots < operando_n >)$$

Donde:

- 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 programa implementado, al que llamaremos `evaluar`, debe conformar la siguiente especificación, al ser invocado desde la línea de comandos:

```
$ evaluar [-h]
```

El parámetro opcional `-h` debe mostrar texto de ayuda por pantalla y terminar. El texto consiste de un breve resumen del propósito del programa, junto con una reseña de las diversas opciones disponibles. En caso de ingresar parámetros erróneos, se debe mostrar un mensaje indicando el error, y a continuación el texto de ayuda.

### 3.1. Consideraciones para comisiones de 3 integrantes

Además de lo detallado hasta el momento, las comisiones integradas por 3 alumnos, deberán cumplir con los siguientes requisitos y funciones

- Contar con opciones de línea de comandos para aceptar tanto la entrada como la salida a través de archivos:

  `$ evaluar [-h] [-i <archivo_entrada>] [-o <archivo_salida>]`

  - Si se especifica un archivo de salida, debe escribirse el resultado en dicho archivo. En caso contrario, debe mostrarse por pantalla:

    `$ evaluar -o archivo_salida`

    En este caso el archivo `archivo_salida` contendrá el resultado.
  - Si se especifica un archivo de entrada, debe leerse la expresión aritmética desde el mismo. En caso contrario debe leerse por entrada estándar.

    `$ evaluar -i archivo_entrada`

    En este caso el archivo `archivo_entrada` contiene la expresión a evaluar.
- Poder evaluar números negativos.
- Gestionar adecuadamente condiciones de overflow y división por cero.
- Implementar la operación negación, con un único operando. Por ejemplo:

  `(- 53)` resulta $-53$

  `(- (* 10 66))` se evalúa como `-(10 * 66)` y da como resultado `-660`.

  Si la operación de negación contiene parámetros insuficientes o demasiados, debe mostrar un mensaje de error, y finalizar la ejecución con *exit status* `OPND_INSUF` u `OPND_DEMAS` respectivamente.

# Sobre la implementación

- Los archivos fuente principales se deben denominar **pila.c, lista.c y evaluar.c** respectivamente.

  En el caso de las librerías, también se deben adjuntar los respectivos archivos de encabezados **pila.h y lista.h**, los cuales han de ser incluidos en los archivos fuente de los programas que hagan uso de las mismas.
- Es importante que durante la implementación del proyecto se haga un uso cuidadoso y eficiente de la memoria, tanto para la reservar (`malloc`), como para liberar (`free`) el espacio asociado a variables y estructuras.
- Se deben respetar con exactitud los nombres de tipos y encabezados de funciones especificados en el enunciado. *Los proyectos que no cumplan esta condición quedarán automáticamente desaprobados.*
- La compilación debe realizarse con el *flag* `-Wall` habilitado. El código debe compilar **sin advertencias** de ningún tipo.

# Sobre el estilo de programación

- El código implementado debe reflejar la aplicación de las técnicas de programación modular estudiadas a lo largo de la carrera.
- En el código, entre eficiencia y claridad, se debe optar por la claridad. Toda decisión en este sentido debe constar en la documentación que acompaña al programa implementado.
- El código debe estar *indentado* y **adecuadamente comentado**.

# Sobre la documentación

*Los proyectos que no incluyan documentación estarán automáticamente desaprobados.*
La documentación debe:

- Estar dirigida a usuarios finales y desarrolladores.
- Explicar detalladamente los programas realizados, incluyendo el diseño de la aplicación y el modelo de datos utilizado, así como toda decisión de diseño tomada, y toda observación que se considere pertinente.
- Incluir explicación de todas las funciones implementadas, indicando su prototipo y el uso de los parámetros de entrada y de salida (tanto en dentro del código fuente como en la documentación del proyecto).
- En general se deberán respetar todas las consignas indicadas en la "Guía para la documentación de proyectos de software" entregada por la cátedra.

# Sobre la entrega

- Las comisiones deben estar conformadas por 2 ó 3 alumnos registrados con la cátedra. La fecha límite para informar a la cátedra quiénes son los integrantes de cada comisión es el **jueves 22 de septiembre de 2016**. Las comisiones de 3 integrantes deberán implementar adicionalmente las consideraciones para comisiones de 3 integrantes.

  *No se aceptarán cambios en los integrantes posteriores a esta fecha.*

  *No se aceptarán trabajos presentados individualmente, por lo tanto estarán desaprobados.*

- El código fuente (**sólo archivos ".c" y ".h"**) y el informe del proyecto deberán ser enviados en un archivo **zip** por mail al asistente de la cátedra hasta las **21:00hs** del día **lunes 10 de octubre de 2016**. Tanto el asunto del mail, como el nombre del archivo comprimido deben respetar el siguiente formato:

  *OC2016_Proyecto_1_-Apellido1-Apellido2-Apellido3*

  Toda comisión que no cumpla este punto estará automáticamente desaprobada.

- El día **martes 11 de octubre de 2016**, de **14:00 a 16:00hs**, en el aula habitual de clase, se deberá entregar un folio plástico (no se aceptarán carpetas) cerrado con cinta adhesiva, conteniendo los siguientes elementos:

- Una carátula que identifique claramente al proyecto, cátedra, institución, fecha e integrantes de la comisión.
- Una impresión en **doble faz** de los archivos ".c" y ".h" enviados por mail.
- La documentación del proyecto impresa en **doble faz**.

No se aceptarán discrepancias entre el código fuente impreso y el enviado por mail.

# Sobre la corrección

- La cátedra evaluará tanto el **diseño** e **implementación** como la **documentación** y **presentación** del proyecto.
- Tanto para compilar el proyecto, como para verificar su funcionamiento, se utilizará la máquina virtual "OCUNS" publicada en el sitio Web de la cátedra.

Tabla 1: Códigos de terminación del programa

| ***Exit status*** | **Valor numérico** | **Significado** |
|---|---|---|
| `EXITO` | `EXIT_SUCCESS` | Terminación exitosa. |
| `EXP_MALF` | 2 | Expresión mal formada, falta o exceso de "`( )`". |
| `LST_NO_INI` | 3 | Lista sin inicializar. |
| `LST_POS_INV` | 4 | Intento de acceso a posición inválida en lista. |
| `OPND_DEMAS` | 5 | Demasiados operandos para el operador. |
| `OPND_INSUF` | 6 | Insuficientes operandos para el operador. |
| `OPND_INV` | 7 | Operando inválido. |
| `OPRD_INV` | 8 | Operador desconocido o inválido. |
| `PIL_NO_INI` | 9 | Pila sin inicializar. |
| `PIL_VACIA` | 10 | La pila se encuentra vacía. |