PROYECTO

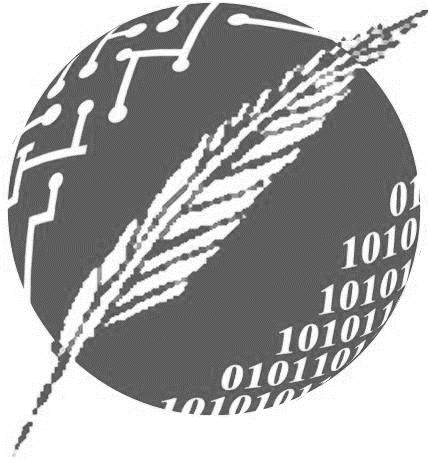
DE

PROGRAMACION EN LENGUAJE C

Profesor: Mg. Alejandro G. Stankevicius

Asistente: Lic. José H. Moyano

UNIVERSIDAD NACIONAL DEL SUR

2/11/2016

Gonzalo Facundo – Miguel Tomas

Este programa permite evaluar expresiones aritméticas ingresadas en preorden.

Para esto se utilizó el lenguaje de programación C y se utilizaron dos TDA, una lista y una pila.

Las expresiones aritméticas utilizan la siguiente sintaxis:

(< operador > < operando1 > < operando2 > . . . < operandoN >)

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)

El programa al ser invocado desde la línea de comandos: **$ evaluar [-h]** muestra un texto de ayuda por pantalla

El programa cuenta con múltiples funciones y subrutinas que serán enumeradas y explicadas brevemente a continuación:

* función evaluar (): Esta función se encarga de capturar la expresión ingresada por el usuario, guardarla en un arreglo y manipularla de forma que se pueda detectar algún error de sintaxis o en su defecto realizar los cálculos correspondientes y devolver un entero.

Las sentencias que pueden llevar a una salida inesperada de la función son:

* Una expresión con paréntesis mal balanceados
* Un número no valido (negativo, decimal, etc)
* Algún operador no es valido
* La cantidad de operandos es insuficiente para aplicar el operador
* Si el operador es / (división) o – (resta), y la cantidad de operadores es mayor a 2
* Función imprimirAyuda(): Esta función al ingresarse una expresión no valida muestra un texto de ayuda con todas las condiciones necesarias para la correcta evaluación de la expresión.
* función suma (operandos): La función suma se invoca cuando se detecta el operador de suma en la pila, recibe como parámetro la lista con los operandos, devolviendo un entero con el resultado de la suma de los elementos de la lista o error si la cantidad de operandos es menor a dos
* función resta (operandos): La función resta se invoca cuando se detecta el operador de resta en la pila, recibe como parámetro la lista con los operandos, la cual debe tener 2 elementos, devolviendo un entero con la resta entre los elementos de dicha lista o un error si la lista es distinta de dos operandos o el resultado es negativo.
* función división (operandos): La función división se invoca cuando se detecta el operador de división en la pila, recibe como parámetro la lista con los operandos, la cual debe tener 2 elementos, devolviendo un entero con la división entre los elementos de dicha lista o un error si la lista tiene más de dos operandos o se realiza una división por cero.
* función multiplicación (operandos): La función multiplicación se invoca cuando se detecta el operador de multiplicación en la pila, recibe como parámetro la lista con los operandos, devolviendo un entero con el resultado del producto de los elementos de la lista o si la lista contiene menos de dos operandos.
* función esOperando (c): Dicha función se encarga de corroborar que ‘c’ sea un operador valido, devolviendo 0 de ser así, 1 en caso contrario.

También se utilizaron las siguientes librerías:

* string.h : Para poder comparar, modificar y trabajar con cadenas de caracteres.
* ctype.h : Se utilizaron los métodos “isBlank()” para distinguir entre un blanco y un carácter e “isDigit()” para validar que el carácter sea un digito.
* lista.h : Creada por el desarrollador, contiene la lista de operandos necesarios para realizar los cálculos.
* pila.h : Creada por el desarrollador, contiene los caracteres apilados ingresados por el usuario en la expresión aritmética
* errores.h : Creada por el desarrollador, contiene una lista de errores declarados por el desarrollador con sus respectivos códigos.

Y los siguientes TDAs previamente mencionados con las siguientes funciones:

TDA PILA:

* pila\_crear (): Inicializa la pila, reserva el espacio en memoria
* pila\_vacia (pila): Devuelve 0 si la pila esta vacía, 1 en caso contrario. Error si la pila no está inicializada.
* apilar (\*pila, \*str): Asigna en el tope de la pila el nuevo elemento ‘str’, devuelve 0 si hay memoria disponible y se realizó el apilado, 1 en caso contrario. Error si la pila no está inicializada.
* tope (pila): Devuelve el elemento en el tope de la pila, error si la pila esta vacía o no inicializada.
* desapilar (\*pila): Elimina el elemento en el tope de la pila y lo retorna, error si la pila esta vacía o no inicializada.

TDA LISTA:

* lista\_crear (): Crea y reserva espacio para una nueva lista de elementos, luego la retorna.
* lista\_cantidad (lista): Devuelve la cantidad de elementos de la lista, error si la lista no está creada.
* lista\_insertar (lista, pos, elem): Inserta ‘elem’ en la posición ‘pos’ de la lista y devuelve 0, si la posición recibida es invalidad devuelve 1.
* lista\_obtener (lista, pos) : Devuelve el elemento de la lista que se encuentra en la posición ‘pos’, error si la posición recibida es invalida.
* Lista\_eliminar (lista, pos): Elimina de lista el elemento en la posición ‘pos’ y de ser necesario reacomoda la lista. Error si la posición es invalida o la lista no está inicializada.
* lista\_adjuntar (lista, elem): Agrega ‘elem’ al final de lista y devuelve 0, en caso de error devuelve 1. Error si la lista no está inicializada.
* lista\_destruir (\*lista): Elimina lista, libera el espacio en memoria y retorna 0. Error si la lista no está inicializada.