UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Trabajo Práctico Autómatas

MATERIA: Sintaxis Y Semántica de los Lenguajes.

PROFESORA: Roxana María Leituz.

COMISIÓN: K2055.

ALUMNOS: Aylén Marta Chazarreta, Brisa Nelly Calzado, Dante Silva, Enrique Antonio Marques y Lisandro Corrales Cespedes.

FECHA DE ENTREGA: 1/10/2024.

Bibliotecas y Macro:

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>
#include <stdlib.h>

#define MAX 100
```

Punto 1:

- **Prototipos de las Funciones**

```c
//--------------------PROTOTIPOS DEL PUNTO 1----------------------
int caracteresPertenecenAlAlfabeto (char*cadena, char*caracteresValidos);
int esPalabra(char* cadena, int* tipoNumero, int modo);
int columna(char c, int quePunto);
```

- **Main**

```c
//--------------------PUNTO 1----------------------
char cadena[MAX];

printf("Ingrese la cadena de numeros separados por '#': ");
scanf("%s", cadena);
char caracteresValidos1[27] = "ABCDEFXabcdefx0123456789-+#";

if(!caracteresPertenecenAlAlfabeto(cadena, caracteresValidos1))
{
    printf("Hay caracteres que no pertenecen al alfabeto.");
    return 0;
}
```

Nosotros elegimos que la cadena ingrese por línea de comando. En primer lugar, verificamos mediante la función "*caracteresPertenecenAlAlfabeto( )*" que todos los caracteres pertenezcan a un conjunto válido de caracteres, definido en "*caracteresValidos1*".

Se declaran los caracteres válidos que son:

- Las letras de la A a la F (mayúsculas y minúsculas) para representar números hexadecimales.
- Los dígitos del 0 al 9 para números decimales, octales y hexadecimales.
- El carácter x/X para marcar números hexadecimales.

- Los símbolos + y - para números decimales, y # como separador entre diferentes tipos de números.

En caso de no verificar se devuelve por consola "Hay caracteres que no pertenecen al alfabeto", y terminamos la ejecución del programa. Si todos los caracteres son válidos, el programa continúa su ejecución.

```
int i=0, j=0, inicio=0;
int contadorDecimales=0, contadorOctales=0, contadorHexadecimales=0;
char subcadena[MAX];

while(cadena[i] != '\0')
{
    if(cadena[i] == '#')
    {
        int largo = i - inicio;
        for(j=0; j<largo; j++)
        {
            subcadena[j] = cadena[inicio + j];
        }
        subcadena[largo] = '\0'; //"cierro" la subcadena
```

A continuación, comenzamos a recorrer la cadena carácter por carácter. Mientras que no se lea fin de cadena (\0), cada vez que lee un “#” tomamos todo lo que hemos leído hasta ese punto y lo copiamos en una subcadena.

```
int tipoNumero = 0;
esPalabra(subcadena, &tipoNumero, 0);

        if(tipoNumero == 1)
        {
            contadorDecimales++;
        } else if(tipoNumero == 2)
        {
            contadorOctales++;
        } else if(tipoNumero == 3)
        {
            contadorHexadecimales++;
        } else if(tipoNumero == 4)
        {
            printf("Hay error lexico en: %s\n", subcadena);
            return 0;
        }

        inicio = i + 1; // proximo caracter + 1
    }
    i++;
}
```

Luego clasificamos y contamos los diferentes tipos de número que hay en la subcadena, los cuales están separados por el caracter “#”. Para hacer esto, utilizamos la función “*esPalabra( )*”, que recibe la subcadena, una referencia a una variable “*tipoNumero*” que indica qué tipo de número estamos procesando: decimal, octal, hexadecimal o si hay un error léxico (como por ejemplo, un número octal que contiene un dígito inválido como '9') y el valor 0 que indica que se está utilizando esta función con las condiciones para el punto 1.

Dependiendo del valor de *“tipoNumero”*, utilizamos una estructura if para incrementar el contador correspondiente (decimales, octales o hexadecimales), o si es un error léxico, mostramos un mensaje de error y terminamos el programa. Finalmente ajustamos la posición de inicio para apuntar al siguiente carácter, de modo que podamos procesar la próxima subcadena cuando encontremos otro “#”.

```
//para procesar el ultimo token (que no termina con #)
int largo = i - inicio;
for(j=0; j<largo; j++)
{
    subcadena[j] = cadena[inicio + j];
}
subcadena[largo] = '\0'; //"cierro" la subcadena

//meter subcadena en el automata, fijarme que tipo de numero es, y sumar
int tipoNumero = 0;
esPalabra(subcadena, &tipoNumero, 0);

if(tipoNumero == 1)
{
    contadorDecimales++;
} else if(tipoNumero == 2)
{
    contadorOctales++;
} else if(tipoNumero == 3)
{
    contadorHexadecimales++;

 } else if(tipoNumero == 4)
 {
     printf("Hay error lexico en: %s\n", subcadena);
     return 0;
 }
```

Finalmente, fuera del bucle, repetimos el mismo proceso para el último token de la cadena. Este último número no tiene un “#” al final, por lo que no se procesa dentro del ciclo.

```
printf("Cantidad de decimales: %d\n", contadorDecimales);
printf("Cantidad de octales: %d\n", contadorOctales);
printf("Cantidad de hexadecimales: %d\n", contadorHexadecimales);
```

Y por último,mostramos en pantalla la cantidad de cada tipo de número que había en la cadena ingresada en un inicio por consola.

- **Funciones**

```
int caracteresPertenecenAlAlfabeto(char* cadena, char*caracteresValidos)
{
    int i = 0;
    int j = 0;

    for (i = 0; cadena[i]; i++)
    {
        int esValido = 0;

        for (j = 0; caracteresValidos[j]; j++)
        {
            if (cadena[i] == caracteresValidos[j])
            {
                esValido = 1;
            }
        }

        if (!esValido)
        {
            return 0;
        }
    }
    return 1; // Todos los caracteres son válidos
}
```

Esta función verifica que todos los caracteres ingresados en la cadena formen parte del alfabeto. La cadena entra por consola.

Entrada: La función recibe un puntero a una cadena de caracteres (char* cadena) y una cadena con los caracteres válidos (char* caracteresValidos).

Salida: Devuelve 1 si todos los caracteres de la cadena pertenecen al conjunto de caracteres válidos; de lo contrario, devuelve 0 si se encuentra algún caracter que no es válido.

Iteración sobre la cadena de entrada: Un bucle for recorre cada carácter de la cadena. Para cada carácter:

- Se inicializa una variable “*esValido*” a 0, indicando que aún no se ha encontrado si el carácter es válido.
- Luego, otro bucle for recorre el arreglo de caracteres válidos, comparando el carácter actual de la cadena con cada uno de los caracteres válidos.
- Si el carácter de la cadena coincide con alguno de los caracteres válidos, se marca como válido (se asigna 1 a esValido).

Validación: Al salir del segundo for, se verifica el valor de esValido:

- Si “*esValido*” sigue siendo 0 (lo que significa que no se encontró una coincidencia con los caracteres válidos), la función retorna inmediatamente 0, indicando que al menos un carácter no pertenece al conjunto válido.
- Si todos los caracteres pasan la validación, al terminar el bucle la función retorna 1, indicando que todos los caracteres son válidos.

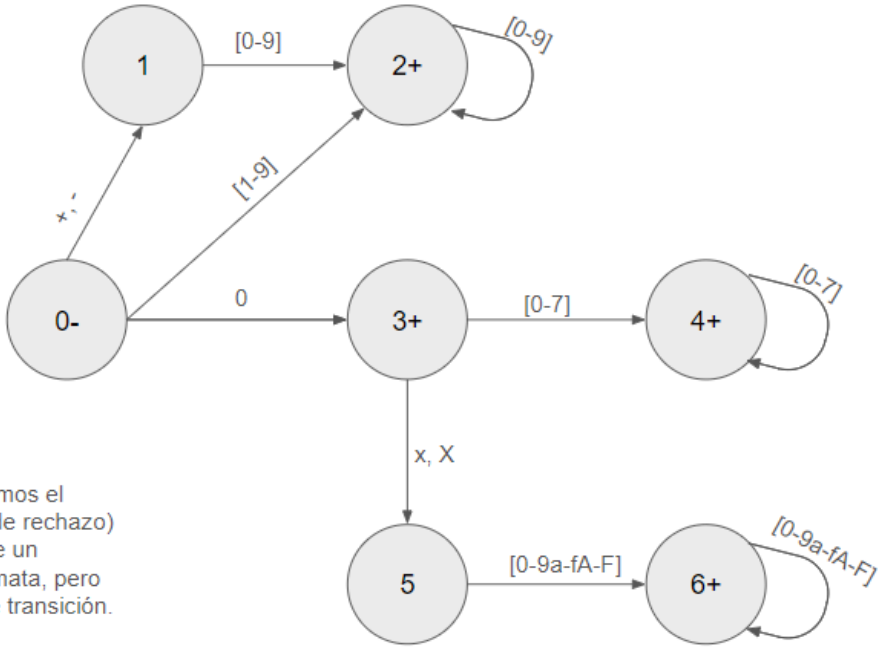
```
int esPalabra(char* cadena, int* tipoNumero, int modo) {
    // Definir tablas de transición combinadas
    // Modo 0: Punto 1 - Números (decimal, octal, hexadecimal)
    // Modo 1: Punto 3 - Expresiones aritméticas (+, -, *, /)

    // Matriz para el autómata del punto 1 (manejo de números)
    static int ttP1[8][7] = {
        {1, 1, 3, 7, 2, 2, 7},
        {7, 7, 2, 7, 2, 2, 7},
        {7, 7, 2, 7, 2, 2, 7},
        {7, 7, 4, 5, 4, 7, 7},
        {7, 7, 4, 7, 4, 7, 7},
        {7, 7, 6, 7, 6, 6, 6},
        {7, 7, 6, 7, 6, 6, 6},
        {7, 7, 7, 7, 7, 7, 7}
    };

    // Matriz para el autómata del punto 3 (manejo de expresiones aritméticas)
    static int ttP3[5][2] = {
        {1, 4},
        {1, 2},
        {3, 4},
        {3, 2},
        {4, 4}
    };
```

```
    int estado = 0;
    int i = 0;
    int c = cadena[0];

    while (c != '\0') {
        // Elegir la tabla y la columna según el modo

        if (modo == 0) {  // Modo números (Punto 1)
            estado = ttP1[estado][columna(c, 1)];

        } else if (modo == 1) {  // Modo expresiones (Punto 3)
            estado = ttP3[estado][columna(c, 0)];
        }
        c = cadena[++i];
    }

    // Verificar estados de aceptación según el modo
    if (modo == 0) {  // Validación para números
        if (estado == 2 || estado == 3) {
            *tipoNumero = 1;  // Decimal
            return 1;
        }
        if (estado == 4) {
            *tipoNumero = 2;  // Octal
            return 1;
        }
        }
        if (estado == 6) {
            *tipoNumero = 3;  // Hexadecimal
            return 1;
        }
        *tipoNumero = 4;  // Error léxico
        return 0;

    } else if (modo == 1) {  // Validación para expresiones aritméticas

        if (estado == 1 || estado == 3) {
            return 1;  // Expresión válida
        }

        return 0;  // Expresión inválida
    }

    return 0;  // No válida en ningún modo
}
```

Para hacer esta función usamos de base lo explicado en clase. Lo único que hicimos fue adaptarlo a nuestro autómata.

Como la función la utilizamos tanto para el punto 1 como para el punto 3 hicimos que a la misma ingrese el *"modo"* que indica en qué punto se está trabajando. El punto 1 se indica con el número 0 y el punto 3 con el número 1.

Para el punto 1, nosotros decidimos hacer un solo autómata junto con su tabla de transición que define los distintos tipos de números. El autómata que hicimos fue el siguiente:

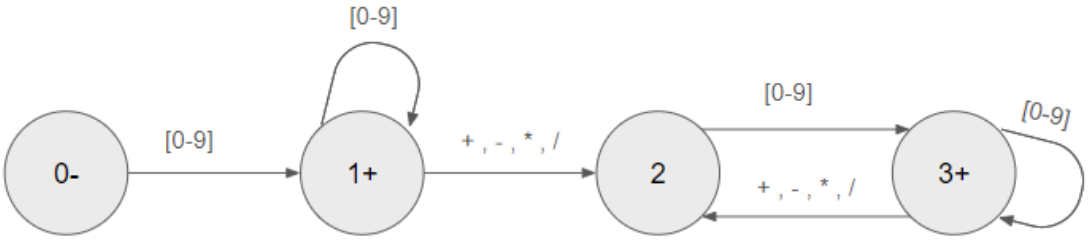


NOTA: No graficamos el estado 7 (estado de rechazo) para que no quede un enchastre el autómata, pero esta en la tabla de transición.

	+	-	0	x/X	[1-7]	[8-9]	[a-fA-F]
0-	1	1	3	7	2	2	7
1	7	7	2	7	2	2	7
2+	7	7	2	7	2	2	7
3+	7	7	4	5	4	7	7
4+	7	7	4	7	4	7	7
5	7	7	6	7	6	6	6
6+	7	7	6	7	6	6	6
7	7	7	7	7	7	7	7

Como se puede ver en el autómata, si el estado de aceptación es el 2, entonces es un número decimal, si es el 3 es un número decimal ya que es el número 0 solo, si es el 4 es un número octal y si es el 6 es un número hexadecimal. En el caso que el estado de aceptación no sea ninguno de esos, significa que había un error léxico en la cadena.

Asimismo, esta función tiene una segunda tabla de transición que pertenece al autómata utilizado en el punto 3. Decidimos hacer un segundo autómata ya que el del punto 1 no tiene en consideración que hayan símbolos aritméticos entre los números, solo el + y el - para indicar si es un número positivo o negativo. En este autómata los estados de aceptación son el 1 y el 3.



NOTA: No graficamos el estado 4 (estado de rechazo) para que no quede un enchastre el autómata, pero esta en la tabla de transición.

	[0-9]	-, +, *, /
0-	1	4
1+	1	2
2	3	4
3+	3	2
4	4	4

```
int columna(char c, int quePunto)
{
    if (quePunto) {
        // Lógica para el análisis de números (punto 1)
        if (c == '+' || c == '-')return 0;   // Signos
        if (c == '0') return 2;               // Prefijo para octal/hexadecimal
        if (c >= '1' && c <= '7') return 4;   // Números octales (1-7)
        if (c == '8' || c == '9') return 5;   // Números decimales
        if (c == 'x' || c == 'X') return 3;   // Prefijo hexadecimal
        if (c == 'a' || c == 'b' || c == 'c' || c == 'd' || c == 'e' || c == 'f' ||
            c == 'A' || c == 'B' || c == 'C' || c == 'D' || c == 'E' || c == 'F')
        {
            return 6;  // Dígitos hexadecimales
        }
        return -1;  // Otros caracteres no válidos

    } else {

        // Lógica para el análisis de expresiones matemáticas (punto 3)
        if (c >= '0' && c <= '9') {
        return 0;
        }
           // Dígitos
        if (c == '+' || c == '-' || c == '*' || c == '/') {
        return 1;

        }
        // Operadores matemáticos
        return -1;  // Otros caracteres no válidos
    }
}
```

En el caso de esta función también tomamos de base lo que hicimos en clase solo que lo ajustamos a nuestra tabla de transición.En el primer condicional evaluamos en qué punto del ejercicio estamos trabajando.Si estamos en el ejercicio 1 (“*quePunto”* = 1 → Verdadero), entramos a unos if secuenciales que nos dicen en qué columna de la tabla de transición quedamos luego de consumir determinado carácter. Si el carácter no pertenece a ninguna de las categorías definidas, retorna “-1” para indicar un valor inválido.

En el otro caso (“*quePunto*” /= 1 → Falso) determina en qué columna de la tabla de transición quedamos luego de consumir determinado carácter pero para el ejercicio 3, si el carácter no pertenece a ninguna de las categorías definidas , retorna “-1” para indicar que son caracteres no válidos.

- **Consola**

```
Ingrese la cadena de numeros separados por '#': 0542#1245#-987#0XaB313cE
Cantidad de decimales: 2
Cantidad de octales: 1
Cantidad de hexadecimales: 1
```

```
Ingrese la cadena de numeros separados por '#': -655#+545#03154
Cantidad de decimales: 2
Cantidad de octales: 1
Cantidad de hexadecimales: 0
```

```
Ingrese la cadena de numeros separados por '#': hola que tal
Hay caracteres que no pertenecen al alfabeto.
```

```
Ingrese la cadena de numeros separados por '#': 0X2144KJ
Hay caracteres que no pertenecen al alfabeto.
--------------------------------
```

```
Ingrese la cadena de numeros separados por '#': 1224Ad
Hay error lexico en: 1224Ad
```

```
Ingrese la cadena de numeros separados por '#': 0241549
Hay error lexico en: 0241549
```

Punto 2:

- **Prototipos de las Funciones**

```c
//--------------------PROTOTIPOS DEL PUNTO 2-----------------------
int convertirCadenaAEntero(char cadena[]);
```

- **Main**

```c
//--------------------PUNTO 2-----------------------

char cadena2[11] ; //por la longitud del int que es 10 + 1 digito
printf("Ingrese una cadena: ");
scanf("%s", cadena2);
int numero = convertirCadenaAEntero(cadena2);
printf("El numero convertido es: %d\n", numero);  // Salida: -1234
```

Se declara la *"cadena2"* de longitud 11 que se va a utilizar para almacenar la cadena escrita por el usuario. Decidimos esa longitud debido a que en C un número entero puede tener hasta 10 dígitos en su representación decimal y hay que tener en cuenta al carácter nulo (\0), en el hipotético caso de que queramos de que sea de

mayor tamaño va a ser indistinto porque int sigue siendo limitada a 10 dígitos y estaríamos usando memoria de más.

Luego el usuario interactúa con el programa insertando la cadena mencionada anteriormente y esta se utiliza como parámetro de la función "*convertirCadenaEntero( )*" que lo que retorna se le asigna a la variable entera "*numero*" para luego imprimirlo.

- **Funciones**

```
//--------------------FUNCIONES DEL PUNTO 2----------------------
int convertirCadenaAEntero(char cadena[]) {
    int i = 0, num = 0, signo = 1;

    // Ignorar espacios en blanco iniciales
    for (i = 0; isspace(cadena[i]); i++);

    // Determinar si hay un signo negativo o positivo
    if (cadena[i] == '-') {
        signo = -1;
        i++;
    } else if (cadena[i] == '+') {
        i++;
    }

    // Convertir los caracteres numéricos en el número entero

     for (num = 0; isdigit(cadena[i]); i++){
        num = 10 * num + (cadena[i] - '0');
     }

    /*El num en 10*num aumenta según el primer numero de la lista
    Ej en el numero 567, num = 5, dsp en la siguiente iteración va a ser 10*5 + 6, y dsp 10*56 + 7
    En (cadena[i] - '0') lo que hace es restar el valor ascii del indice de la cadena y del 0
    Por ejemplo 0 = 48 ascii, si el numero que quiero es 1 = 49 ascii,
    hago 49 - 48 y ahí me da el 1 del que en verdad quiero*/

    // Devolver el número convertido, aplicando el signo
    return signo * num;
}
```

En esta función se convierte una cadena de caracteres numéricos en un entero.

La función comienza inicializando la variable " *i* " que es el índice que vamos a utilizar para recorrer la cadena de caracteres, la variable "*num*" que es la variable que acumula el valor entero resultante de la conversión y la variable "*signo*" para indicar la positividad o negatividad del número.

Después se realiza un ciclo for que por medio de la función *isspace( )* (función que verifica si un carácter dado es un espacio en blanco y pertenece a <ctype.h>) se ignoran los espacios en blanco que tiene la cadena.

Una vez omitido los espacios en blanco se entra en un estructura de condición que determina si es positivo o negativo y en función de eso se modifica o no la variable signo y avanzando una posición en la cadena.

Luego se realiza el último ciclo for en el que se verifica si el carácter dentro de esa posición de la cadena es un dígito decimal(función *isdigit( )* perteneciente a <ctype.h>) y de acuerdo a la posición en la que se encuentra se le agrega un peso representando los órdenes de magnitud para después sumarla a la variable "*num".*

Por último , se retorna el producto entre la variable "*num"* y la variable "*signo".*

- **Consola**

```
Ingrese una cadena: -5455
El numero convertido es: -5455
```

```
Ingrese una cadena: +2445
El numero convertido es: 2445
```

```
Ingrese una cadena: 32452
El numero convertido es: 32452
```

Punto 3:

- **Prototipos de las Funciones**

```
//--------------------PROTOTIPOS DEL PUNTO 3-----------------------
int esOperador(char c);
int precedencia(char op);
int hacerOperacion(int a, int b, char op);
int evaluarCadenaPosfija(char* tokens);
void infijaAPosfija(char* expresion, char* cadenaDeSalida);
```

- **Main**

```
    //--------------------PUNTO 3----------------------
    char expresion[MAX];
    printf("Ingrese un calculo sencillo:");
    scanf("%s", expresion);
    char caracteresValidos3[15] = "0123456789-+*/";

    if(!caracteresPertenecenAlAlfabeto(expresion, caracteresValidos3)) // si NO verifica algun caracter
    {
        printf("La expresion contiene caracteres que no pertenecen al alfabeto.\n");
        return 0;
    }

    if (!esPalabra(expresion, NULL, 1)) // si NO verifica expresion
    {
        printf("La expresion contiene un error sintactico\n");
        return 0;
    }
    char expresionPosfija[MAX];
    infijaAPosfija(expresion, expresionPosfija);
    printf("Resultado: %d\n", evaluarCadenaPosfija(expresionPosfija));

    return 0;
}
```

Para este punto, decidimos que la expresión para realizar sea ingresada por consola. Definimos un vector *"expresion"* para almacenar el cálculo que desea realizar el usuario. Esta expresión será verificada por las funciones del punto 1, adaptadas al nuevo alfabeto, y validando que sea un cálculo simple con un nuevo AF.

Se declaran los caracteres válidos que son:

- Los dígitos del 0 al 9.
- Los símbolos +, -, * y /.

También, definimos un vector auxiliar del mismo tamaño que nos ayudará para almacenar la misma expresión pero en otra notación.

Luego, se ejecuta el procedimiento *"infijaAPosfija(expresion, expresionPosfija)"*. Con él, se busca pasar de una notación infija a posfija, por eso se pasan por parámetro los vectores mencionados anteriormente.

Finalmente, se realiza el cálculo de la expresión desde la notación posfija mediante la función *"evaluarCadenaPosfija(expresionPosfija)"* y se muestra por pantalla el resultado.

- **Funciones**

```
int esOperador(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}
```

Una función auxiliar sencilla que se utilizará para validar si el caracter evaluado es o no un operador, retornando 1 si lo es y 0 si no lo es.

```
int precedencia(char op) {
    switch (op) {
        case '+':
        case '-':
            return 1;
        case '*':
        case '/':
            return 2;
        default:
            return 0;
    }
}
```

Otra función auxiliar que nos devuelve un valor que representa la precedencia del operador leído. Resultará de utilidad para evaluar qué debe realizar la función que arma la cadena con la expresión en notación postfija.

```
int hacerOperacion(int a, int b, char op) {
    switch (op) {
        case '+': return a + b;
        case '-': return a - b;
        case '*': return a * b;
        case '/': return a / b;
    }
    return 0;
}
```

La última función auxiliar, recibirá dos operandos enteros y un char con el operador correspondiente. Devuelve el resultado de la operación como entero.

```
void infijaAPosfija(char* expresion, char* cadenaDeSalida) {
    char stack[MAX];
    int top = -1, k = 0;
    int len = strlen(expresion);
    int i = 0;

    for (i = 0; i<len; i++) {
        if (isdigit(expresion[i])) {
            while (isdigit(expresion[i])) {
                cadenaDeSalida[k++] = expresion[i++];
            }
            cadenaDeSalida[k++] = ' ';
            i--;
        } else if (esOperador(expresion[i])) {
            while (top != -1 && precedencia(stack[top]) >= precedencia(expresion[i])) {
                cadenaDeSalida[k++] = stack[top--];
                cadenaDeSalida[k++] = ' ';
            }
            stack[++top] = expresion[i];
        }
    }
    while (top != -1) {
        cadenaDeSalida[k++] = stack[top--];
        cadenaDeSalida[k++] = ' ';
    }
    cadenaDeSalida[k] = '\0';
}
```

La función "*infijaAPosfija( )"* recibe como argumentos la cadena con la expresión a calcular ingresada por consola (en notación común o infija) y otra cadena auxiliar donde se guardará la expresión anterior pero en notación postfija.

Contiene un arreglo de chars "*stack"* que se utilizará como pila, para ir guardando y sacando los operadores leídos de acuerdo a su precedencia. Int "*top"* será el puntero a la cima de nuestra pila, "*k"* e **"***i"* serán utilizados como índices. En "*len"* guardaremos el largo de la expresión infija ingresada como argumento, lo utilizaremos como criterio de corte para la iteración principal.

Una vez dentro del *for*, si se lee un caracter decimal (y hasta que dejen de leerse decimales) los iremos guardando dentro de la "*cadenaDeSalida"*. Cuando esto ocurra insertamos un espacio y decrementamos el índice del *for*, para no perdernos el caracter operador que cortó el *while* para dígitos.

Si dejamos de leer dígitos, necesariamente debe ser porque leímos un operador, ya que la cadena fue previamente verificada por la función "*caracteresPertenecenAlAlfabeto"* y **"***esPalabra"*. En este caso se evalúa que nuestra pila no esté vacía y si el operador que tenemos en la pila tiene mayor precedencia que el que leímos, lo pasamos a la cadena postfija e insertamos un espacio. Caso contrario, guardamos el operador leído en la pila.

Cuando concluye el for insertamos en la cadena postfija los operadores que hayan quedado en la pila.

```c
int evaluarCadenaPosfija(char* tokens) {
    int stack[MAX], top = -1;
    char numStr[MAX];
    int numStrIndex = 0;
    int len = strlen(tokens);
    int i = 0;

    for (i = 0; i<len; i++) {
        if (isdigit(tokens[i]) || (tokens[i] == '-' && isdigit(tokens[i+1]))) {
            numStr[numStrIndex++] = tokens[i];
            if (!isdigit(tokens[i+1])) {
                numStr[numStrIndex] = '\0';
                stack[++top] = convertirCadenaAEntero(numStr);
                numStrIndex = 0;
            }
        } else if (esOperador(tokens[i])) {
            int val2 = stack[top--];
            int val1 = stack[top--];
            stack[++top] = hacerOperacion(val1, val2, tokens[i]);
        }
    }
    return stack[top];
}
```

Esta función se encarga de evaluar una expresión matemática escrita en notación postfija. Recibe como parámetro un puntero a una cadena de caracteres que representa la expresión.

Se declara un arreglo "*stack*" que utilizamos como pila y una variable de tipo entero para señalar la cima. También, utilizamos un arreglo de caracteres temporal para manejar números de varios dígitos en la cadena, y con ella, el índice "*numStr*" para construir el número. *"Len"* e "*i*" nos ayudan a establecer la longitud de la cadena "*tokens*" y el índice para recorrerla.

Dentro del for, recorremos la cadena *"tokens"* y evaluamos si son números (positivos o negativos) o si se trata de un operador:

En el caso de ser un número, se utiliza el vector auxiliar "*numStr*" para construir un número de varios dígitos. Se verifica si el siguiente carácter no es un dígito, de ser así, se añade '\0' para finalizar la composición de ese número. Luego se convierte el número, que hasta ahora estaba como cadena de caracteres a un tipo de dato numérico utilizando la función "*convertirCadenaAEntero*" del punto 2, y se empuja a la pila de números.

En el caso de que se trate de un operador, se sacan 2 valores de la pila de números, se realiza la operación con la función "*hacerOperacion*" y el resultado se vuelve a empujar en la pila.

Por último, luego de procesar todos los caracteres de la cadena original, el resultado se encontrará en la parte superior de la pila "*Stack[top]*", y este valor es el que se retorna. 123

- **Consola**

```
Ingrese un calculo sencillo: 672/7-985
Resultado: -889
```

```
Ingrese un calculo sencillo: 214*74/6+4
Resultado: 2643
```

```
Ingrese un calculo sencillo: 965*4+5/9
Resultado: 3860
```

Consola de los tres puntos juntos:

```
Ingrese la cadena de numeros separados por '#': 125#65#021#0x1Ab
Cantidad de decimales: 2
Cantidad de octales: 1
Cantidad de hexadecimales: 1
Ingrese una cadena: 326
El numero convertido es: 326
Ingrese un calculo sencillo: 24+56/2+4*8
Resultado: 84
```

```
Ingrese la cadena de numeros separados por '#': 0#032#0XF
Cantidad de decimales: 1
Cantidad de octales: 1
Cantidad de hexadecimales: 1
Ingrese una cadena: -56
El numero convertido es: -56
Ingrese un calculo sencillo: 98*36/2+5
Resultado: 1769
```

Bibliografía:

- Kernighan, B. W., & Ritchie, D. M. (1988). *El lenguaje de programación C* (2da ed.). Pearson Educación.
- Muchnik, J., & Díaz Bott, A. M. (2012). *Sintaxis y semántica de los lenguajes* (2da ed.). UTN FRBA CEIT.