TP1: Manejo de archivos y cadenas

El trabajo práctico número 1 tiene fecha de entrega para el día **15/10**, y está divido en tres partes:

1. una serie de funciones para manejo de cadenas *(strutil.c)*
2. una calculadora en notación posfija *(dc.c)*
3. un conversor de notación infija a notación posfija *(infix.c)*

Contenido

Manejo de cadenas

Se pide implementar las funciones de manejo que cadenas que se describen a continuación. Adjunto en el sitio de descargas se puede encontrar un archivo *strutil.h* con todos sus prototipos.

Para la implementación de estas funciones no se permite el uso de TDAs. Sí se permite, no obstante, el uso de las funciones de la biblioteca estándar de C string.h (excepto `strtok` y `strstr`). Se recomiendan, en particular:

- `strlen`, `strchr`
- `strcpy`/`strncpy`
- `strdup`/`strndup`
- `snprintf`

Se dsescriben a continuación las cuatro funciones a implementar.

substr()

La función `substr()` permite obtener un prefijo de longitud k de una cadena dada.

Por ejemplo, la llamada `substr("Hola mundo", 6)` devolvería la cadena `"Hola m"`. El resultado debe ser una nueva cadena de memoria dinámica con dichos caracteres (y, por supuesto, el carácter de fin de cadena). Ejemplos de uso:

```
substr("Hola mundo", 6)         → "Hola m"
substr("Algoritmos", 30)        → "Algoritmos"
substr("", 2)                   → ""

const char* ejemplo = "Ejemplo";
substr(ejemplo, 2)              → "Ej"
substr(ejemplo + 4, 2)          → "pl"
```

Complejidad algorítmica: se espera que la función tenga complejidad $\mathcal{O}(k)$.

split()

La función `split()` divide una cadena en subcadenas, con una división por cada ocurrencia de un caracter de separación determinado. Por ejemplo, separando por comas:

```
split("abc,def,ghi", ',')  →  ["abc", "def", "ghi"]
```

En C, devolveremos el resultado como un arreglo dinámico de cadenas dinámicas terminado en `NULL`. Esto es:

```c
// Ejemplo de arreglo dinámico de cadenas
char **strv = malloc(sizeof(char*) * 4);
strv[0] = strdup("abc");
strv[1] = strdup("def");
strv[2] = strdup("ghi");
strv[3] = NULL;
```

Considerar los siguientes casos borde:

```
split("abc,,def", ',')  →  ["abc", "", "def"]
split("abc,def,", ',')  →  ["abc", "def", ""]
split(",abc,def", ',')  →  ["", "abc", "def"]
split("abc", '\0')      →  ["abc"]

split("", ',')  →  [""]
split(",", ',') →  ["", ""]
```

Complejidad algorítmica: se espera que la función tenga complejidad $\mathcal{O}(n)$, siendo n la longitud de la cadena inicial.

join()

La función `join()` es la inversa de `split()`. Concatena un arreglo de cadenas terminado en NULL mediante un caracter de separación:

```c
// Ejemplo de uso de join
char **strv = split("abc,def,ghi", ',');
char *resultado = join(strv, ';');  // "abc;def;ghi"

char **palabras = split("Hola mundo", ' ');
char *otro_resultado = join(palabras, ',');  // "Hola,mundo"
```

Casos borde:

```
join([""], ',')            →  ""
join(["abc"], ',')         →  "abc"
join(["", ""], ',')        →  ","
join([], ',')              →  "" // Join de arreglo vacío, {NULL} en C.
join(["abc", "def"], '\0')  →  "abcdef"
```

Complejidad algorítmica: se espera que la función tenga complejidad $\mathcal{O}(n)$, siendo n la longitud de la cadena resultante.

Las pruebas del corrector automático proveen una indicación del comportamiento de `join()` (si bien **todas las funciones deben correr en tiempo lineal**, `join()` provee una dificultad de implementación mayor a `split()` o `substr()`).

Este sería un test con comportamiento lineal:

```
[ RUN      ] test_join.test_complejidad_10000
[       OK ] test_join.test_complejidad_10000 (7 ms)
[ RUN      ] test_join.test_complejidad_20000
[       OK ] test_join.test_complejidad_20000 (14 ms)
[ RUN      ] test_join.test_complejidad_30000
[       OK ] test_join.test_complejidad_30000 (17 ms)
[ RUN      ] test_join.test_complejidad_40000
[       OK ] test_join.test_complejidad_40000 (27 ms)
[ RUN      ] test_join.test_complejidad_50000
[       OK ] test_join.test_complejidad_50000 (33 ms)
[ RUN      ] test_join.test_complejidad_60000
[       OK ] test_join.test_complejidad_60000 (40 ms)
```

Y este con comportamiento cuadrático:

```
[ RUN      ] test_join.test_complejidad_10000
[       OK ] test_join.test_complejidad_10000 (48 ms)
[ RUN      ] test_join.test_complejidad_20000
[       OK ] test_join.test_complejidad_20000 (618 ms)
[ RUN      ] test_join.test_complejidad_30000
[       OK ] test_join.test_complejidad_30000 (1354 ms)
[ RUN      ] test_join.test_complejidad_40000
[       OK ] test_join.test_complejidad_40000 (2425 ms)
[ RUN      ] test_join.test_complejidad_50000
[       OK ] test_join.test_complejidad_50000 (4019 ms)
[ RUN      ] test_join.test_complejidad_60000
[       OK ] test_join.test_complejidad_60000 (5722 ms)
```

free_strv()

`free_strv()` libera la memoria asociada con un arreglo dinámico de cadenas dinámicas:

Si bien no es obligatorio probar la biblioteca, es muy recomendable hacer pruebas unitarias para comprobar el correcto funcionamiento, en particular para los casos borde.

Calculadora en notación posfija

Se pide implementar un programa `dc` que permita realizar operaciones matemáticas. La calculadora leerá exclusivamente de entrada estándar (no toma argumentos por línea de comantos), interpretando cada línea como una operación en notación polaca inversa (también llamada *notación posfija*, en inglés *reverse Polish notation*); para cada línea, se imprimirá por salida estándar el resultado del cálculo.

Ejemplo de varias operaciones, y su resultado:

```
$ cat oper.txt
5 3 +
5 3 -
5 3 /
3 5 8 + +
3 5 8 + -
3 5 - 8 +
2 2 + +
0 1 ?
1 -1 0 ?
5 sqrt

$ ./dc < oper.txt
8
2
1
16
-10
6
ERROR
ERROR
-1
2
```

Funcionamiento

- Todas las operaciones trabajarán con números enteros, y devolverán números enteros. Se recomienda usar el tipo de C `long` para permitir operaciones de más de 32 bits (p.ej. 3^{3^3}).
    - Si se hace uso de la biblioteca calc_helper mencionada más abajo, el tipo será, simplemente, `calc_num`.
- El conjunto de operadores posibles es: suma (`+`), resta (`-`), multiplicación (`*`), división entera (`/`), raíz cuadrada (`sqrt`), exponenciación (`^`), logaritmo (`log`) en base arbitraria, y operador ternario (`?`).
    - Todos los operadores funcionan con dos operandos, excepto `sqrt` (toma un solo argumento) y el operador ternario (toma tres).
- Tal y como se describe en la bibliografía enlazada, cualquier operación aritmética *a op b* se escribe en postfijo como `a b op` (por ejemplo, $3 - 2$ se escribe en postfijo como `3 2 -`).

Es útil modelar la expresión como una pila cuyo tope es el extremo derecho de la misma (por ejemplo en `3 2 -`, el tope es `-`); entonces, se puede decir que lo primero que se desapila es el operador, y luego los operandos **en orden inverso**.

  - Para operaciones con un solo operando, el formato es obviamente `a op` (por ejemplo, `5 sqrt`). Por su parte, para el operador ternario, el ordenamiento de los argumentos seguiría el mismo principio, transformándose `a ? b : c` en `a b c ?`.

- Ejemplos (nótese que toda la aritmética es entera, y el resultado siempre se trunca):

  - `20 11 -` → `20-11 = 9`
  - `20 -3 /` → `20/-3 = -6`
  - `20 10 ^` → `20^10 = 10240000000000`
  - `60 sqrt` → `√60 = 7`
  - `256 4 ^ 2 log` → $\log_2(256^4) = 32$
  - `1 -1 0 ?` → `1 ? -1 : 0 = -1` (funciona como en C)

Formato de entrada

- Cada línea de la entrada estándar representa una operación en su totalidad (produce un único resultado); y cada una de estas operaciones operación es independiente de las demás.

- Los símbolos en la expresión pueden ser números, u operadores. Todos ellos estarán siempre separados por uno o más espacios; la presencia de múltiples espacios debe tenerse en cuenta a la hora de realizar la separación en tokens.

  - Nota adicional: puede haber también múltiples espacios al comienzo de la línea, antes del primer token; por otra parte, no necesariamente habrá un espacio entre el último token y el caracter salto de línea que le sigue.

  - Todo esto es tenido en cuenta en la función `dc_split` (ya implementada) de la biblioteca calc_helper.

- El resultado final de cada operación debe imprimirse en una sola línea por salida estándar (*stdout*). En caso de error, debe imprimirse —para esa operación— la cadena `ERROR`, *también* por salida estándar, y sin ningún tipo de resultado parcial. Tras cualquier error en una operación, el programa continuará procesando el resto de líneas con normalidad.

- Está permitido, para el cálculo de potencias, raíces y logaritmos, el uso de las funciónes de la biblioteca estándar `<math.h>`.

Condiciones de error

El mensaje `ERROR` debe imprimirse como resultado en cualquiera de las siguientes situaciones:

1. cantidad de operandos insuficiente (`1 +`)

2. al finalizar la evaluación, queda más de un valor en la pila (por ejemplo `1 2 3 +`, o `+ 2 3 -`)

3. **errores propios de cada operación matemática**, descritos a continuación:

   - división por 0
   - raíz de un número negativo
   - base del logaritmo menor a 2
   - potencia con exponente negativo

Conversor desde notación infija

Una vez implementada la calculadora en notación postfija, se desea agregarle soporte para operaciones expresadas en notación infija, es decir, notación aritmética común:

- $4 + 2 - 1$.
- $5/3$.
- $2 * (3 + 7)$.

No obstante, en lugar de modificar el código de `dc`, se pide escribir un segundo programa, `infix`, que actúe como *filtro de conversión*, esto es: leerá de entrada estándar operaciones en notación infija (una por línea), e imprimirá por salida estándar la representación en postfijo de la misma operación. Ejemplo:

```
$ cat arith.txt
3 + 5
5 - 3
8 / 2 + 1
9 - 2 * 4
(9-2) * 4
5 + 4 ^ 3 ^ 2

$ ./infix < arith.txt
3 5 +
5 3 -
8 2 / 1 +
9 2 4 * -
9 2 - 4 *
5 4 3 2 ^ ^ +

$ ./infix < arith.txt | ./dc
8
2
5
1
28
262149
```

Como referencia bibliográfica, la conversión se puede realizar mediante el algoritmo *shunting yard* (ver página de Wikipedia en castellano o en inglés).

Formato de entrada

Cada línea de la entrada consistirá de una secuencia de *tokens*; cada uno de ellos podrá ser:

- uno de los cinco operadores aritméticos `+ - * / ^`
- un paréntesis de apertura, `(`; o de cierre, `)`
- un número entero, no negativo y en base decimal

Se debe aceptar espacios en blanco en cualquier punto de la expresión, excepto entre los dígitos de un valor numérico.

Se garantiza, por último, que todas las expresiones de entrada estarán bien formadas: paréntesis balanceados, espaciado conforme a norma, etc.

Asociatividad y precedencia

Todos los operadores asocian por la izquierda, excepto la exponenciación, `^`, que asocia por la derecha.

Por otra parte, `^` es el operador de mayor precedencia, seguido de `*` y `/` (ambos al mismo nivel); `+` y `-` son, ambos, los operadores de menor precedencia.

Biblioteca calc_helper

En el sitio de descargas se puede obtener, para este TP, dos archivos (`calc_helper.h` y `calc_helper.c`) que conforman una biblioteca de tipos y funciones útiles para la implementación de ambos programas, *dc* e *infix*.

La biblioteca proporciona:

- funciones para procesar texto (`dc_split` e `infix_split`)
- funciones para reconocer elementos válidos en la entrada (`calc_parse`, y sus tipos asociados)
- una pila de enteros, `pilaint_t`, implementada a partir del TDA Pila de punteros genéricos (útil para la versión no recursiva de *dc*)

Los archivos de la biblioteca están extensamente comentados, y en uno de los tipos se explica algunas ampliaciones que se pueden realizar para hacer aún más fácil la implementación de los programas. En particular, se sugiere agregar varios campos en el `struct calc_oper` para que, modificando la función `calc_parse()`, se informe de las distintas propiedades de cada operador.

Criterios de aprobación

El código entregado debe ser claro y legible y ajustarse a las especificaciones de la consigna. Debe compilar sin advertencias y correr sin errores de memoria.

La entrega incluye, obligatoriamente, los siguientes archivos de código:

- *strutil.c* con las implementaciones de las funciones `substr`, `split`, `join` y `free_strv`.

- el código de `dc` en *dc.c* y el de `infix` en *infix.c*
- el código de los TDAs auxiliares requeridos por los programas (incluyendo *.c* y *.h*)
- un archivo *deps.mk* que exprese las dependencias del proyecto en formato makefile. Este archivo deberá contener solamente dos líneas que indiquen, para cada programa, de qué *objetos* depende su ejecutable; por ejemplo:

```
# Ejemplo de archivo deps.mk para el TP1
dc: dc.o pila.o
infix: infix.o strutil.o
```

La entrega se realiza únicamente en forma digital a través del sistema de entregas, con todos los archivos mencionados en un único archivo ZIP.