PROYECTO INTÉRPRETE ARITMÉTICO
Análisis y Diseño de Algoritmos
Universidad Panamericana

Ain Bolaños Cortés
Esteban Eguiarte Maldonado
Mariana Sánchez Esparza

Septiembre 2023

1 Descripción y Análisis del Problema

El siguiente programa tiene como objetivo resolver operaciones aritméticas básicas en las cuales el usuario solo podrá utilizar la suma y el producto. El usuario debe ingresar la operación aritmética válida expresada en una cadena de texto, en la que se permiten paréntesis, el signo de suma (+), el signo de multiplicación (*), números enteros y números decimales. Es importante destacar que no se pueden utilizar paréntesis para expresar el producto.

Uno de los problemas que hemos considerado como equipo es la posibilidad de que se ingresen caracteres no válidos, como letras o signos diferentes a la suma y el producto. Además, hemos tenido en cuenta otros problemas potenciales, como la inserción de dobles puntos decimales, paréntesis que se abren pero no se cierran, paréntesis vacíos y la posibilidad de que no haya suficientes operandos u operadores.

2 Descripción de la Solución Propuesta

La solución propuesta para abordar el problema se basa en una serie de funciones que trabajan en conjunto para resolver operaciones aritméticas a partir de una cadena de texto ingresada por el usuario.

Primero, la función `juntar` se encarga de eliminar espacios innecesarios y validar la entrada, asegurando que solo se trabaje con caracteres válidos. A continuación, se tiene la función `calcular`, que permite implementar recursividad para reducir cada expresión aritmética a expresiones mucho más simples y operables. Esta función comienza por utilizar `eliminar_parentesis`, que elimina paréntesis redundantes y simplifica la expresión. Luego, `num_valido` verifica si los números presentes en la cadena son válidos, evitando problemas como múltiples puntos decimales o caracteres no numéricos.

La función `encontrar_operador` busca operadores en la cadena, asegurándose de que haya suficientes operandos para realizar operaciones válidas. Esto se logra con la ayuda de `validar_par`, que se utiliza para encontrar el paréntesis que cierra el inicialmente identificado. Si se encuentra un operador, se llama a la función `calcular` para realizar los cálculos necesarios.

Finalmente, la función `main` sirve como punto de entrada del programa, donde se recibe la cadena del usuario, se procesa a través de todas las funciones mencionadas y se presenta el resultado. Esta solución proporciona una herramienta completa y precisa para resolver operaciones aritméticas básicas a partir de una interfaz de texto, garantizando la validación de la entrada y el cumplimiento de las reglas de precedencia en las operaciones.

3 Algoritmos

En esta sección se presentará un algoritmo por cada función hecha, cada uno se presentará en el orden según aparezca en el código principal.

3.1 Algoritmo correspondiente a `function(juntar)`

1. Inicializar una cadena vacía llamada `nueva` que contendrá la cadena final.
2. Obtener la longitud de la cadena de entrada y asignarla a la variable `n`.
3. Iniciar un bucle que recorrerá cada caracter de la cadena de entrada desde el priemro hasta el último. Utilizaremos una variable llamada `ind` para llevar un seguimiento de la posición actual.
4. Asignar el caracter en la posición `ind` de la cadena de entrada a la variable `char`.
5. Comprobar si `char` no es un espacio, un salto de línea o un tabulador para determinar si debe incluirse en la cadena `char`. Usaremos los caracteres correspondientes en Julia `('', '̃ y ń')`
6. Concatenar los caracteres que cumplan con la condición de la cadena `nueva`.
7. Después de recorrer todos los caracteres, devolver la cadena `nueva` como resultado.

3.2 Algoritmo correspondiente a `function(num_valido)`

1. Inicializamos una variable llamada `cuenta` con el valor 0. Esta variable se utiliza para verificar si un número contiene más de un punto decimal.
2. Asignamos la longitud de la cadena a la variable `n`.
3. Iteramos a través de cada caracter de la cadena, comenzando desde el primero hasta el último, para verificar si son números, operadores válidos o puntos decimales. Utilizaremos la variable `ind` para recorrer los caracteres.
4. Asignamos el caracter en la posición `ind` de la cadena a la variable `char`.
5. Comprobamos si `char` es una cifra, un operador o un punto decimal. Si ninguno de estos casos se cumple, se considera un error ya que no es una entrada válida para operaciones.
6. Si se encuentra un punto decimal, incrementamos la variable `cuenta` en uno. Al verificar todos los caracteres, si `cuenta` supera 1, se considera un error, ya que un número no puede tener más de un punto decimal.
7. La función debe devolver la entrada siempre y cuando sea un número válido.

3.3 Algortimo correspondiente a `function(eliminar_parentesis)`

1. Asignamos la longitud de la cadena a la variable `n`.
2. Si la longitud es 0, se considera un error, ya que hay un paréntesis vacío. Esto puede ocurrir cuando la función `calcular` llama a esta función y la cadena comienza con un paréntesis.
3. Si lo anterior no se cumple y el primer caracter es un parentesis, llamamos a la función `validar_par` para determinar el índice del paréntesis que cierra el paréntesis inicial. El resultado se asigna a la variable `ind_final`.
4. Si `ind_final` es igual al último índice, significa que el primer el último paréntesis son redundantes, por lo que devolvemos la cadena de texto desde el segundo caracter hasta el penúltimo caracter. De esta mandera, eliminamos los paréntesis redundantes.
5. Llamamos recursivamente a la función `eliminar_parentesis` en el resultado final, ya que no hay paréntesis redundantes en ese punto. El resultado se asigna a la variable `texto`.
6. La función devuelve la variable `texto` ya que no hay paréntesis redundantes en la cadena original.

3.4 Algoritmo correspondiente a `function(validar_par)`

1. Inicializamos la variable `ind` con el valor 2. Esta variable se utilizará para iterar y comparar todos los caracteres, comenzando desde el segundo caracter, ya que el primero se sabe que es un paréntesis.
2. Asignamos la longitud de la cadena a la variable `n`.
3. Iniciamos un contador que servirá para mantener un seguimiento de la cantidad de paréntesis abiertos. El contador comienza en uno, y se incrementa en uno cuando se encuentra un paréntesis abierto, y se resta uno cuando se encuentra un paréntesus cerrado.
4. Mientras el índice sea menor o igual al último, realizamos comparaziones para verificar si los paréntesis están completamente emparejados.
5. Asignamos el valor del caracter actual a la variable `char`, y el valor del caracter anterior se asigna a la variable `char_anterior`.
6. Si el caracter actual es un paréntesis abierto y el caracter anterior pertenece a `cifras`, se considera un error, ya que no se puede multiplicar con un paréntesis, Lo mismo ocurre si después de un paréntesis cerrado, hay otro paréntesis abierto. Al final, se sumo uno al contador para indicar que se ha encontrado un paréntesis abierto.
7. Si el caracter actual es un paréntesis cerrado y el caracter inmediatamente anterior es un paréntesis abierto o un operador, se considera un error. No puede haber paréntesis vacíos ni operadores junto a un paréntesis cerrado. Además, si el caracter siguiente pertenece a `cifras`, también se considera un error, ya que no se puede multiplicar con un paréntesis.
8. Restamos uno al contador por encontrar un paréntesis cerrado.
9. Si el contador llega a ser 0 y hemos llegado al último caracter, retornamos el valor actual del índice.
10. Si el caracter siguiente inmediato no es un paréntesis cerrado, también retornamos `ind` para continuacr la comparación.
11. Al finalizar la revisión, sumamos uno al índice para continuar la comparación y repetir el proceso.
12. Si el contador es diferente de 0 después de revisar todos los caracteres, se considera un error.

3.5 Algoritmo correspondiente a `function(encontrar_operador)`

1. Recibe dos argumentos: el caracter y la cadena.
2. Inicializa la variable `i` con el valor 1, que represetará el índice del caracter que se va a revisar y ayudará a verificar si se cumple la condición.
3. Asigna a la variable `n` la cantidad de caracteres en la cadena.
4. Mientras el caracter actual sea menor o igual al último caracter, se utiliza para separar paréntesis y encontrar operadores.
5. Asigna a la variable `char_act` el caracter actual.
6. Si el caracter actual es un paréntesis abierto, actualiza la variable `i` con el índice del paréntesisi que cierra ese paréntesis. Este resultado proviene de la función `validar_par`.
7. Si el valor de `char_act` es igual al caracter que recibió la función del inicio, la función debe devolver la posición en la que se encuentra. Esto ocurre cuando se encuentra un operador.
8. Si no se cumple la condición anterior, la función devuelve 0, ya que la posición 0 no es válida y continuará buscando el siguiente operador.

3.6 Algoritmo correspondiente a `function(calcular)`

1. Recibimos la cadena como entrada.
2. Asignamos a la variable `n` la cantidad de caracterers en la cadena.
3. Si el primer caracter de la cadena es un paréntesis abierto, actualizamos la variable `texto` con el resultado de la función `eliminar_parentesis`.
4. Ejecutamos la función `num_valido`. Si el resultado de la función es verdadero, entonces es un número válido y podemos aplicar la función `parsec` para leer el número.
5. Si no es un número válido, significa que hemos encontrado un operador, por lo que ejecutamos la función `encontrar_operador`, que inicialmente busca el operador de suma.
6. Si la función `encontrar_operador` indica que el operador está en la primera posición (índice 1), es un error, ya que no hay operadores suficientes.
7. Si el operador no está en la priemra posición pero sí en la última, también es un error, porque falta el último operador.
8. Si el índice no es igual a 1, entonces hay suficientes operandos, por lo que procedemos a separar y llamamos nuevamente a la función `calcular` de manera recursiva para buscar más operadores o eliminar paréntesis.
9. Si la función `encontrar_operador` devuelve el valor 0, que no existe, significa que no se encontró el operador de suma. En este punto, la función buscará el operador de multiplicación.
10. Ejecutamos nuevamente la función `encontrar_operador`, pero esta vez, buscamos el operador de multiplicación.
11. Si la función `encontrar_operador` indica que el operador está en la primera posición (índice 1), es un error, ya que no hay operandos suficientes.
12. Si el operador no está en la primera posición pero sí en la última también es un error porque falta el último operador.
13. Si el índice no es igual a 1, entonces hay suficientes operandos, por lo que procedemos a separar y llamamos nuevamente a la función `calcular` de manera recursiva para buscar más operadores o eliminar paréntesis.
14. Ahora que todo está separado, podemos realizar las operaciones primero en los paréntesis, luego en el producto y finalmente en la suma.

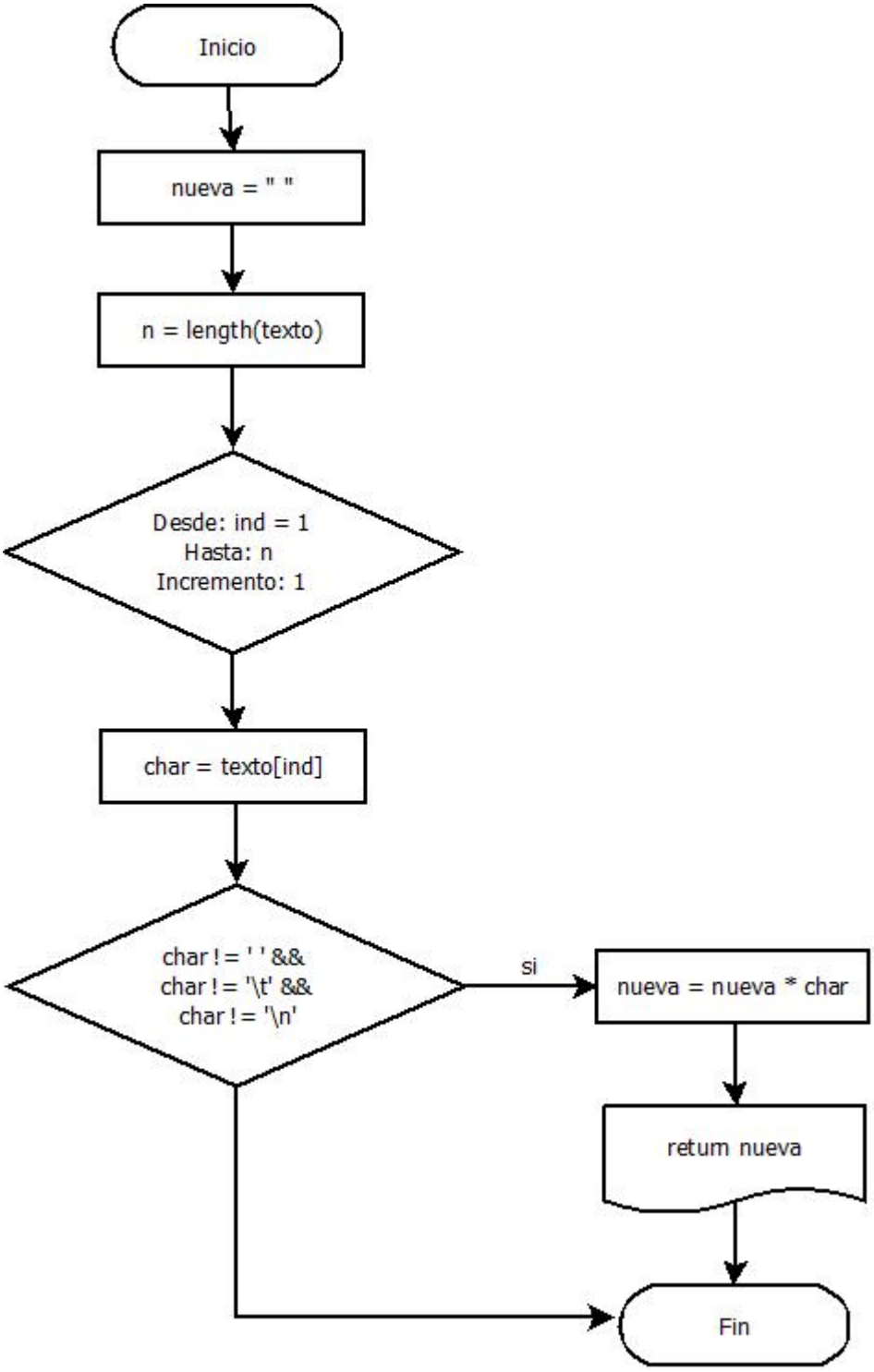
3.7 Algoritmo correspondiente a `function(main)`

1. Creamos una variable `fin` con el valor `true`.
2. Creamos una cadena vacía llamada `entrada` para recibir la cadea del usuario.
3. Solicitamos al usuario que ingrese una cadena y proporcionamos onstrucciones sobre cómo funciona el programa.
4. Utilizamos un bucle `while` para leer la cadena ingresada por el usuario.
5. Usamos el comando `readline` para capturar la entrada del usuario y concatenarla a la variable `entrada`, que inicialmente estaba vacía. Continuamos concatenando elementos hasta que la cadena ingresada por el usuario esté vacía.
6. Actualizamos la variable `fin` a `false` para interrumpir el ciclo `while`.
7. Aplicamos la función `juntar` a la cadena ingresada.
8. Si el resultado después de aplicar `juntar` es una cadena vacía, se produce un error, ya que se entregó una cadena vacía.
9. Mostramos el resultado de la función `calcular`.

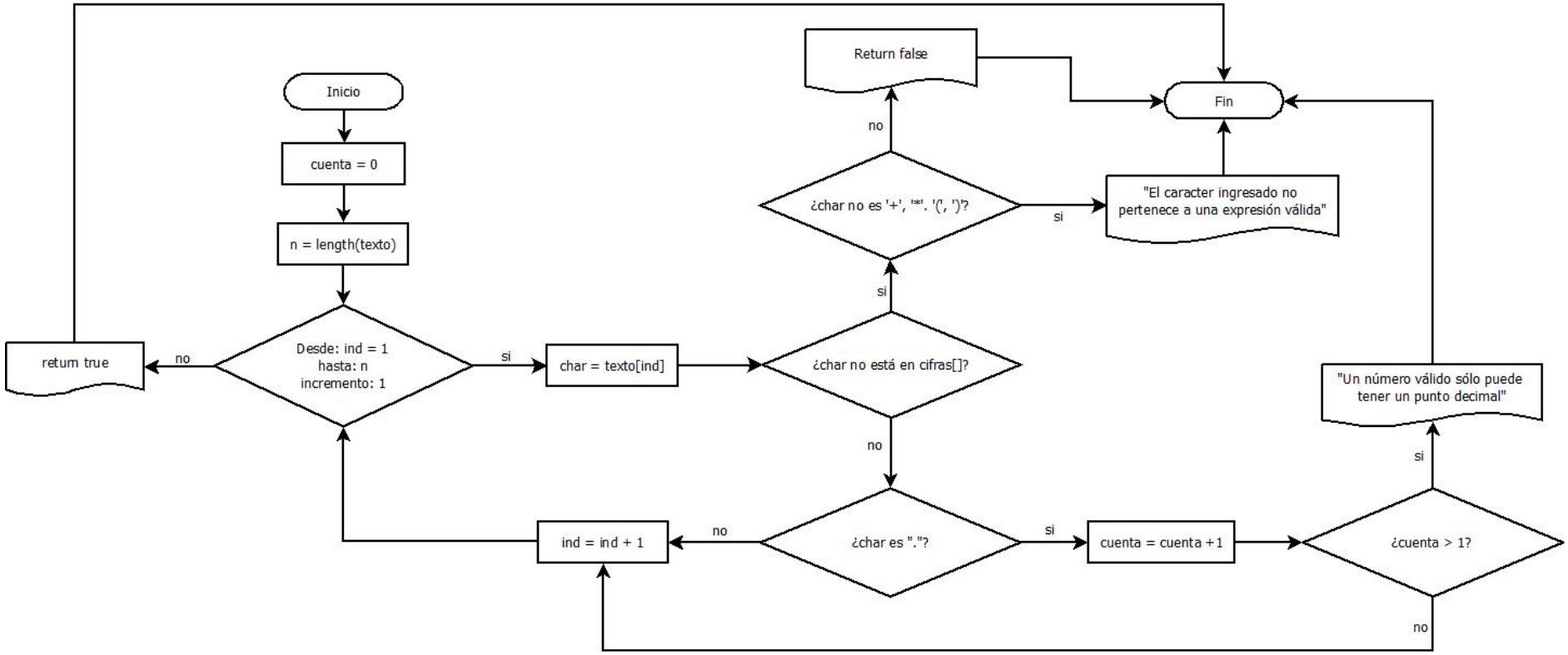
4 Diagramas de Flujo

En esta sección se presentará un diagrama de flujo por cada función hecha, cada uno se presentará en el orden según aparezca en el código principal.

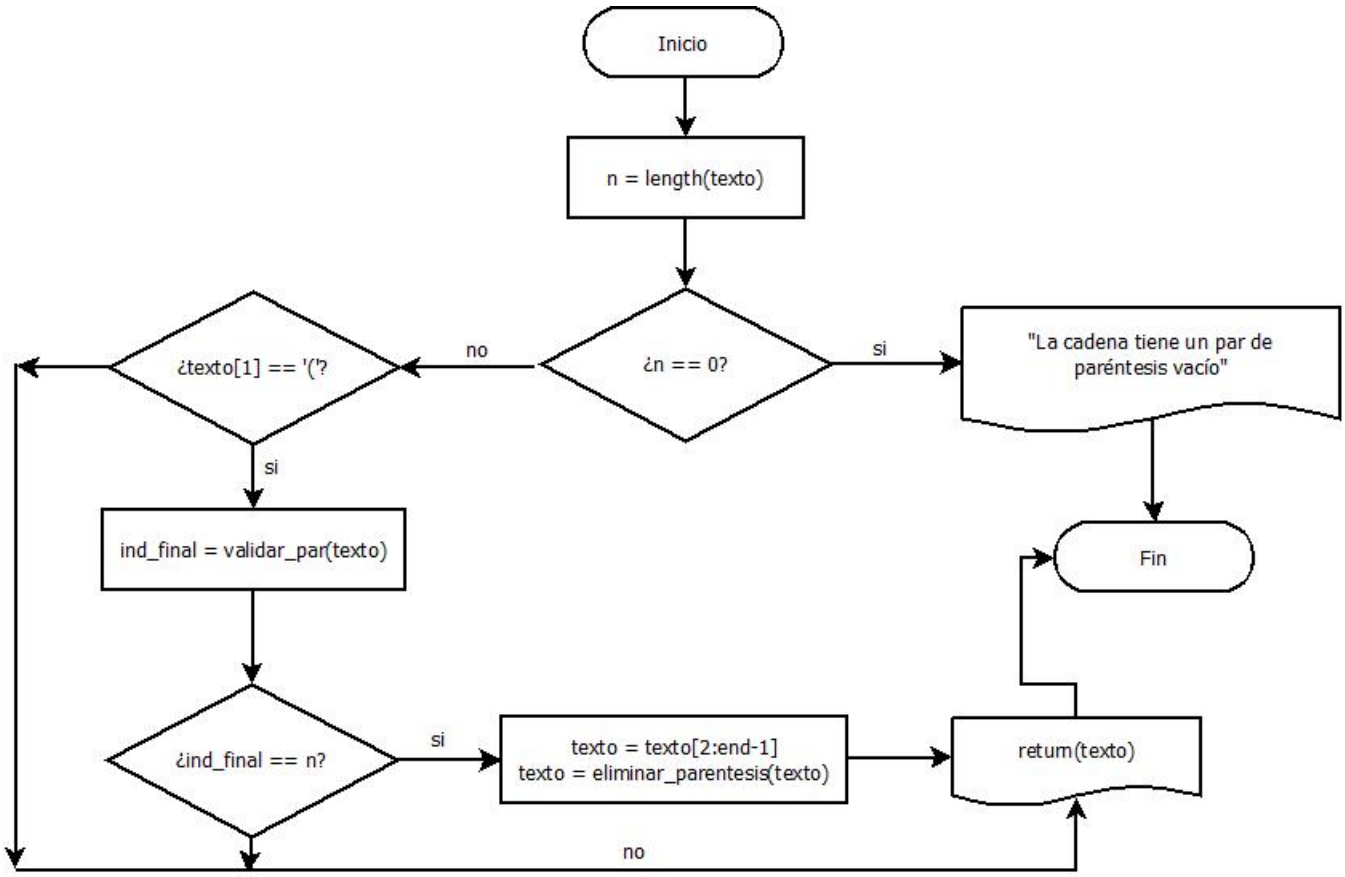
4.1 Diagrama de flujo correspondiente a `function(juntar)`



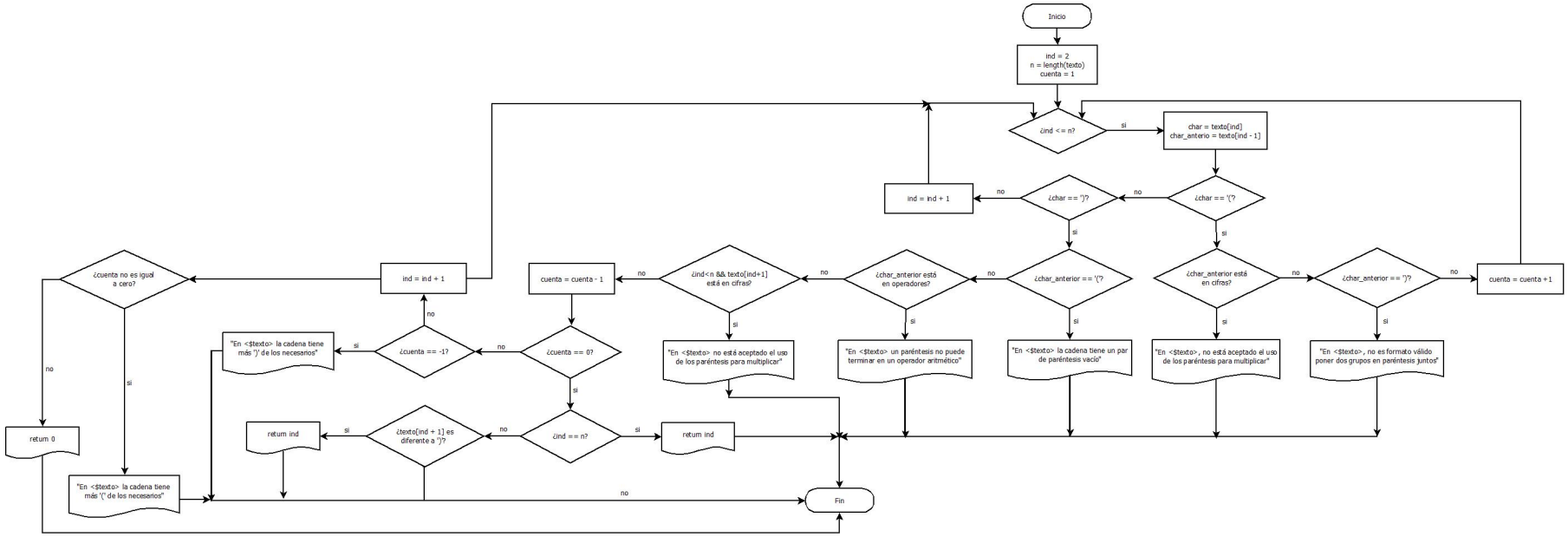
4.2 Diagrama de flujo correspondiente a `function(num_valido)`



4.3 Diagrama de flujo correspondiente a `function(eliminar_paréntesis)`

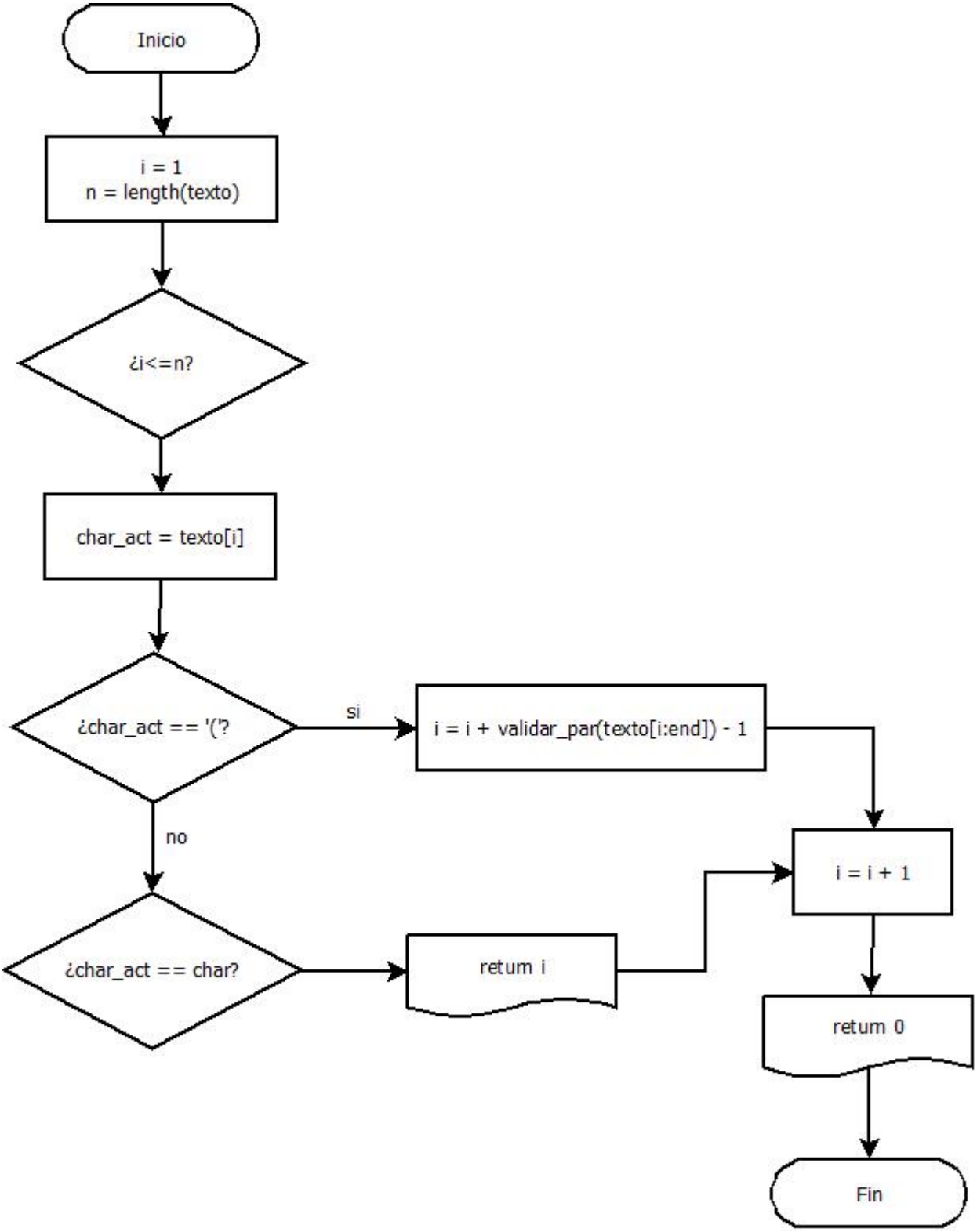


4.4 Diagrama de flujo correspondiente a `function(validar_par)`

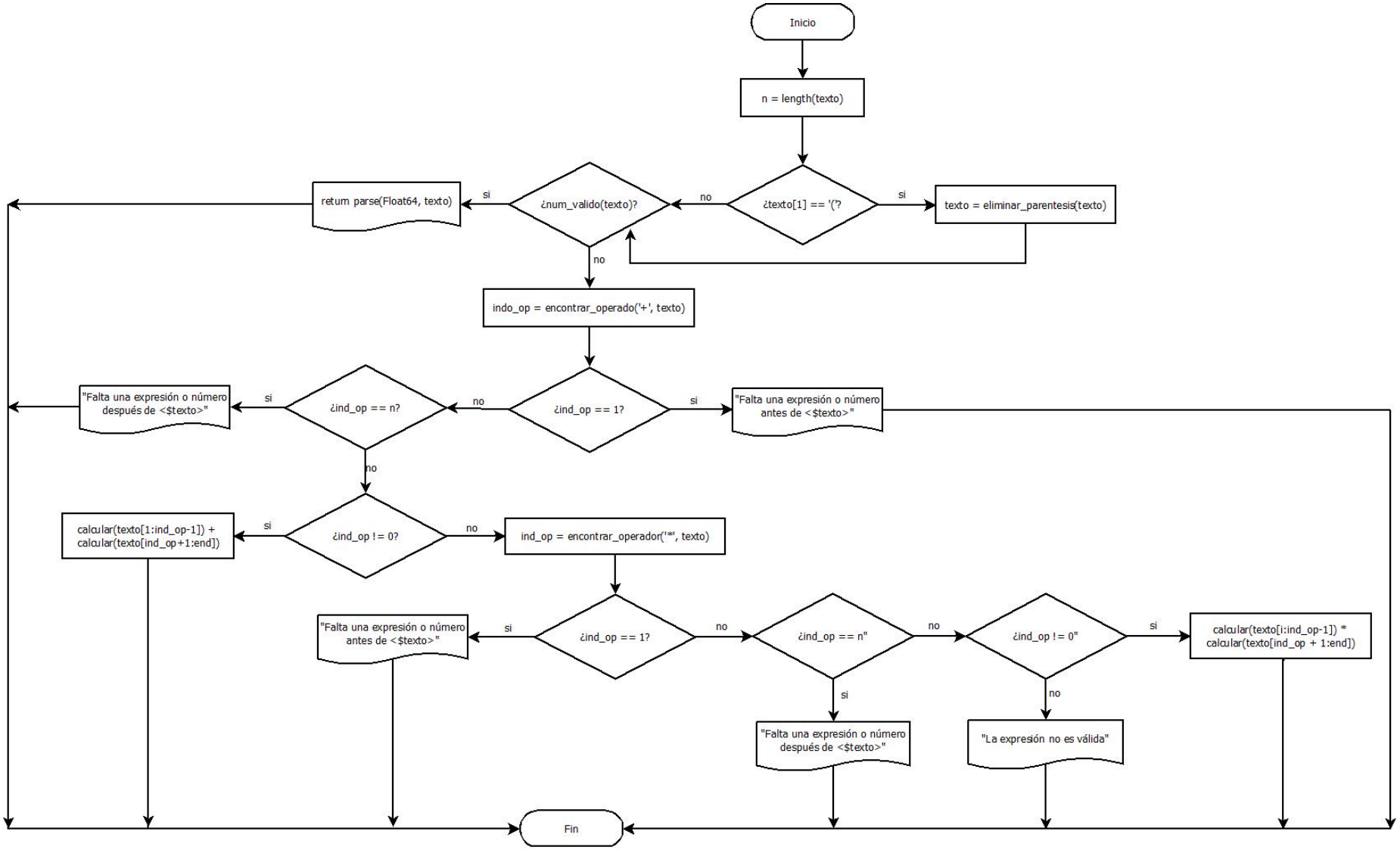


∞

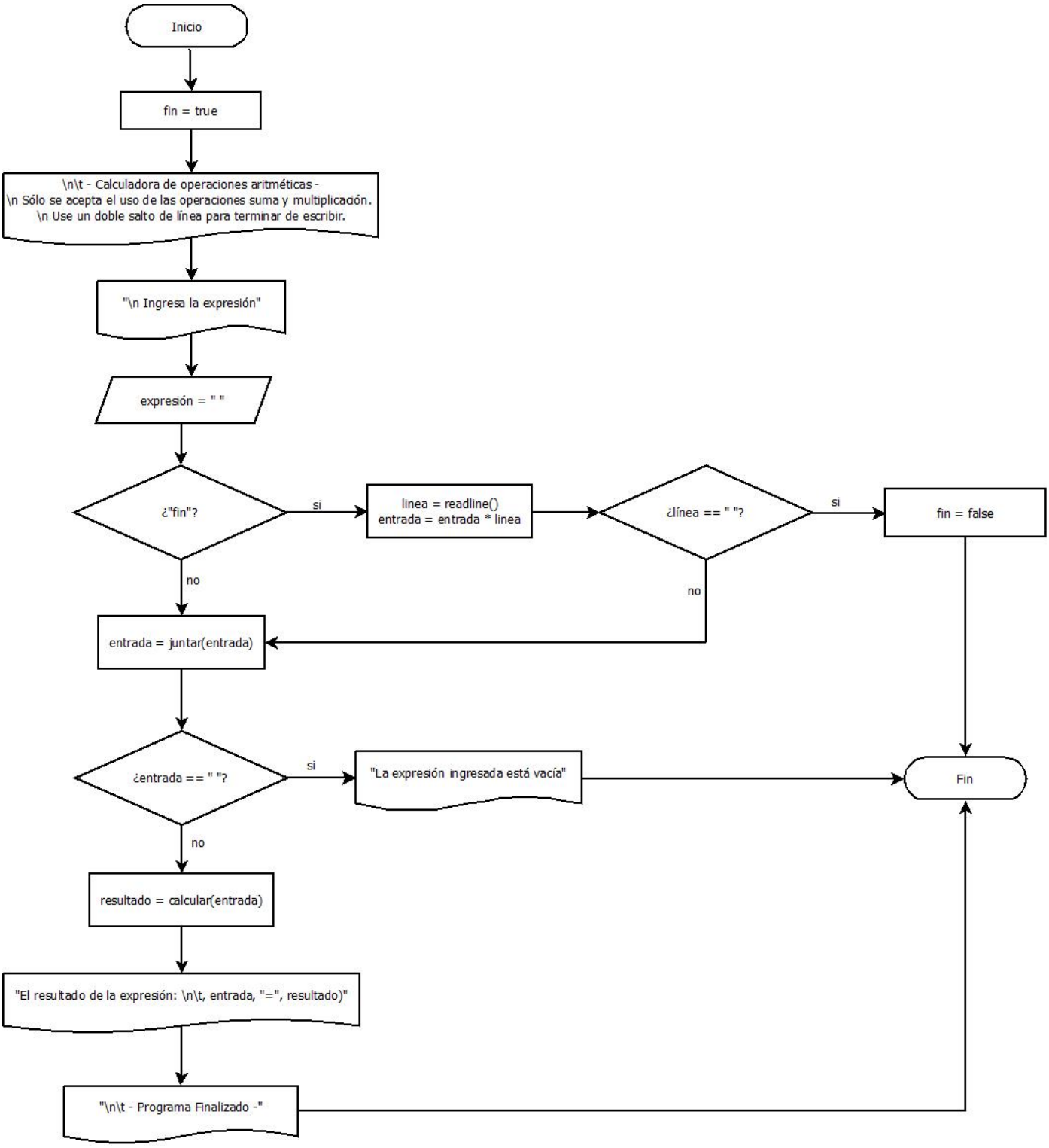
4.5 Diagrama de flujo correspondiete a `function(encontrar_operador)`



4.6 Diagrama de flujo correspondiente a `function(calcular)`



4.7 Diagrama de flujo correspondiente a `function(main)`



5 Instrucciones para utilizar el programa

Para utilizar el interpretador aritmético, el usuario deberá seguir los siguientes pasos en su terminal:

1. El usuario deberá navegar a la carpeta que contiene los archivos siguientes: `interpretador_aritmético.jl` e `interprete_aritmetico_funciones.jl`.
2. Una vez que se esté dentro de la carpeta, el usuario deberá ejecutar el comando `julia` para abrir la terminal de Julia.
3. Deberá utilizar el comando `include("interpretador_aritmetico.jl")`. Esto ejecutará la función del menú, que dará acceso a la función `calcular`.
4. Dentro del menú, el usuario podrá ingresar una expresión aritmética. Esta expresión puede contener los siguientes símbolos: '(', ')', '+', '*', '.' y los números indo-arábigos.
5. Es importante mencionar que el usuario podrá añadir saltos de línea, espacios y tabuladores en cualquier parte de la expresión para mejorar su legibilidad. Para finalizar la entrada de texto, el usuario deberá utilizar un doble salto de línea.
6. Si la expresión no es válida desde el punto de vista aritmético, la función mostrará un mensaje de error que indicará el lugar en la expresión donde se encuentra el problema y la razón de la falla, para que de esta forma, el usuario tenga la oportunidad de modificar su expresión.

6 Porcentaje de contribución por integrante

Ain Bolaños Cortés - 33.33%
Esteban Eguiarte Maldonado - 33.33%
Mariana Sánchez Esparza - 33.33%