Reporte de Resultados

- **Nombre**: Murillo García Sebastián

- **Práctica #2**: Viernes 9 de abril, 2021

- **Aspectos prácticos adicionales**.

Algunos de los aspectos prácticos que el equipo tuvo que investigar y discutir para tomar decisiones concretas sobre la aplicación del programa y el diseño del código para su funcionamiento fueron los siguientes:

- **Método para convertir en tokens la cadena de expresión**. Como primera propuesta para la "tokenización" de la cadena, uno de los integrantes del equipo sugirió utilizar el método **split(sep, maxsplit)** propio de las cadenas de caracteres en Python. Inicialmente pareció ser la mejor idea para esta tarea, pues parecía ser la más sencilla, pero tuvo un inconveniente. Una vez que el método necesita de un caracter o de una cadena como parámetro de separación, los integrantes del equipo discutieron sobre el separador indicado y se llegó a la conclusión de que el caracter que representa al espacio (' ') podría funcionar bien; por lo tanto, el usuario tenía que separar los operadores y operandos por medio de un espacio en la consola. Al momento de implementar este criterio para la entrada de la expresión, el equipo llegó al acuerdo de que no era muy cómodo para el usuario la utilización de este criterio. ¿Por qué? Porque en ninguna calculadora, ya sea "física" o digital, el usuario tiene que introducir algún tipo de separador para el funcionamiento. Incluso si pensamos que el usuario utiliza un teclado numérico en su día cotidiano (pensemos por ejemplo en alguien que se dedique a la contaduría), este criterio de separación por medio de espacios no le beneficia ni en lo más mínimo. Por lo tanto, se decidió descartar este método para la *tokenización* de la cadena a pesar de ser el más sencillo para nosotros como programadores.

  Para solucionar este reto también se buscó en internet algún otro método, biblioteca o pista en python, pero no se encontró algo que fuera lo suficientemente entendible para el equipo. Como el equipo no comprendía en su totalidad el porqué de su funcionamiento, bajo qué condiciones iba a arrojar los resultados esperados, si fallaba, etcétera, se decidió hacer un **método propio** para *tokenizar* la cadena de entrada y tener un control sobre esta situación. Este método utiliza caracteres simples para tokenizar los operadores y los paréntesis; rebanadas de la cadena para

tokenizar números completos, variables e incluso el operador de división entera. El método incluso considera y resuelve el caso especial que hace referencia a la utilización de números o variables negativas de la siguiente manera:

-3+7	→	0-3+7
5*(-21/19)	→	5*(0-21/19)

De esta manera, y después de hacer pruebas extensas, el método que *tokeniza* la cadena de entrada cumple con los requerimientos necesarios para que el código programado funcione como lo deseado.

- **Manejo de errores y excepciones**. Si bien este aspecto se investigó y se resolvió rápidamente, no dejó de ser importante para el pleno funcionamiento del programa y se consideró necesario como un "medio de comunicación" dentro del mismo código (específicamente para saber si un token es un número y para manejar los errores relacionados con las pilas para la creación de los árboles de expresión) y con el usuario (para casos relacionados con la validez de la entrada de la expresión en general o de variables insertadas en la misma). De esta manera, al momento de utilizar el programa tanto los programadores como el usuario van a tener una idea más clara de lo que no está funcionando correctamente. Si bien los métodos y clases programados que ocuparon de este aspecto fueron **isFloat(value)**, **EmptyCollectionException(RuntimeError)** y **InvalidExpression(RuntimeError)**, también se investigaron otros errores que aparecieron en la consola al momento de hacer las pruebas. Estos últimos errores hacían referencia a temas propios de Python y matemáticas como la división entre cero (ZeroDivisionError) o cuando una operación genera un número que sobrepasa los límites de soporte *float* (OverflowError: (34, 'result too large')). De esta manera, también pudimos considerar cuándo un error salía de nuestros límites como programadores y cuándo era enteramente nuestra responsabilidad arreglar.

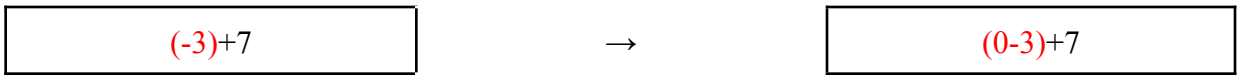
- **Aspectos conceptuales adicionales**.

En relación a aspectos conceptuales que el equipo tuvo que investigar para la creación de la calculadora no se tuvo casi ningún tema. Lo más cercano a un aspecto conceptual en el desarrollo de esta práctica fue lo relacionado con el aspecto matemático; además, este concepto estuvo bastante relacionado con las decisiones en cuanto al diseño del programa. La pregunta es la siguiente: ¿qué pasa si el usuario quiere hacer alguna operación que hiciera referencia a la **raíz** de un número?, ¿tenemos que expresar otro operador que represente a este operador? Para este momento ya se había determinado que los operadores que se iban a utilizar en el programa eran los de adición, sustracción, multiplicación, división y potencia. Si bien pudimos haber agregado un operador más para representar las operaciones con raíz, el equipo abordó el problema desde un aspecto más matemático al hacer uso del operador de potencia que ya estaba considerado dentro del programa. Esto permite que el usuario pueda hacer operaciones más allá de una raíz cuadrada y pueda manejar expresiones un poco más complicadas.

Un inconveniente de esto es que para cualquier caso se tienen que manejar raíces por medio de la potencia a una fracción y si el usuario no conoce de esta herramienta matemática para lograr el objetivo deseado, no podrá aprovechar en su totalidad el operador de potencia. También el equipo tiene claro que este caso es algo adicional a lo solicitado en la práctica, pero aún así el tiempo dio la oportunidad de poder agregar este tipo de posibilidades para el usuario.

Otro aspecto conceptual bastante sencillo que se tuvo que meditar e implementar en el código estuvo relacionado con los números y variables negativas. La pregunta era sencilla: ¿qué se tenía que hacer cuando se quería expresar a un **número negativo** y no a la operación de sustracción? Para entender a qué nos referimos, solo es necesario observar la tabla ejemplo que se puso en el apartado *Aspectos prácticos adicionales*. Al preguntarle al profesor sobre este aspecto se llegó a la conclusión de que este caso a considerar dependía enteramente del criterio del programador. Nos puso el ejemplo de que las calculadoras tienen un botón especial para representar a números negativos y que resolvía perfectamente esta cuestión. El problema se presenta en el momento en que la práctica descarta la interfaz gráfica de la calculadora. Uno de los integrantes sugirió que se podía representar a un número o a una variable negativa por medio del caracter de guión largo (guión corto: ‘-’ , guión largo: ‘—’) previo a este, pero esto tiene una desventaja: ¿qué sucede si el usuario no conoce cuál *shortcut* describe a este caracter? En este caso, si coloca un guión corto y el programa arrojaría un error en el momento de crear al árbol de expresión (se probó y eso sucedía). Además, vamos a hacer sinceros, la mayoría de las ocasiones en la calculadora no hacemos mucho uso de aquel botón para diferenciar a menos que nos marque el característico “Syntax Error” en el display del dispositivo y la calculadora aún puede hacer la

operación que pretendemos hacer. Por lo tanto, la solución a este caso tuvo dos caras: la utilización de paréntesis para especificar que se hará una operación con un número negativo y la adición de un cero antes del al momento de tokenizar la cadena. Veamos rápidamente a qué nos referimos:



Este sencillo movimiento que se hace al tokenizar la cadena facilita cualquier tipo de operación. El aspecto conceptual que utilizamos para resolver este problema se puede resumir en la siguiente oración: “la sustracción al cero del valor absoluto de un número te da como resultado un número negativo”. En palabras más burdas podemos decir lo siguiente: “si al cero le restas un número positivo, te va a dar su número negativo”. Que nos perdonen los grandes matemáticos al describir un número negativo de esta manera. Esta herramienta específicamente nos ayudó para considerar los casos en los cuales se presenta un número negativo al principio de la cadena o cuando se encuentra posterior a un paréntesis izquierdo. Esto también tiene un inconveniente: si el usuario quisiera trabajar con una expresión que tuviera muchos números negativos, la expresión estaría llena de paréntesis por todos lados para que sea válida, pero creemos que esto puede ser un poco más amigable para el usuario en el sentido de que no tiene que ocupar otro tipo de caracter. Además, desde otro punto de vista esto hace que el usuario sea completamente consciente de la estructura de su expresión de entrada en el sentido de que conoce qué cosas están siendo evaluadas.

- **Retos**.

Durante el diseño y desarrollo del código de la práctica, el equipo se enfrentó con diferentes retos en el momento de la unión de las partes asignadas en las subtareas e incluso en el diseño mismo del código. Estos se presentaron desde las decisiones sobre la validez de las cadenas de entrada hasta la evaluación de los árboles de expresión de dichas cadenas. A continuación presentamos algunos de los retos más importantes con los que el equipo tuvo que lidiar:

- ¿Se van a implementar **otros operadores** en el programa? Si la respuesta es sí (que sí lo fue), ¿cómo es que vamos a considerar estos casos? No solo se agregaron más operadores que se tuvieron que considerar en el momento de tokenizar y evaluar la expresión, sino que también se agregó la posibilidad de poder implementar **variables** en la expresión. Esto generó grandes retos para el diseño de los métodos necesarios para el correcto funcionamiento de estos

agregados. Fue incluso un mayor reto debido a que una vez que ya se había creado parte del código, los integrantes tuvieron que trabajar nuevamente e independientemente en sus subtareas para que la novedad del operador o de la variable funcionara y la unión de sus partes fuera de manera armónica. Para ello se tuvo que revisar nuevamente método por método, pues se obtuvieron muchos errores al intentar probar instantáneamente el método general después de agregar los nuevos elementos.

- Otro reto que se presentó fue aquel que tiene que ver enteramente con la **revisión** de la cadena de expresión en su modalidad de tokens. Al momento de discutir sobre cómo es que se iba a resolver esta cuestión, el equipo no llegó a una idea clara de cómo se iba a lograr el objetivo. En la materia de Estructuras de Datos al realizar el proyecto de la calculadora, cada integrante mencionó una manera distinta de cómo abordó el problema. La solución más sencilla y menos elaborada parecía ser la que un integrante mencionó: en la interfaz gráfica no se permitió al usuario ingresar caracteres que consideraron como "prohibidos" en el orden de la expresión. Bajo esta visión fue que el equipo pudo implementar el revisor de la cadena. Si bien se descartó la interfaz gráfica de la calculadora para esta gráfica, lo que se hizo fue un método llamado **casos** que checaba dichos caracteres prohibidos en la cadena.

- **Algunos detalles y criterios a tomar en cuenta**.

A continuación se presentan algunos criterios relacionados con la validez de las entradas y otras cuestiones que se deben de tomar en cuenta:

1. Los operadores válidos en el programa son:

Suma	+
Resta	-
Multiplicación	*
División con decimales	/
División entera	//
Potencia	^

2. Una entrada con espacios es considerada errónea

Error	Corrección
3 * 2 - (40 // 10)	3*2-(40//10)
22 ^ (1 / 4 - 3 / 2) - 10	22^(1/4-3/2)- 0

3. Si se desea poner alguna operación que contenga a un número negativo, es necesario poner paréntesis

Error	Corrección
3+-7	3+(-7)
3--7	3-(-7)
3*-7	3*(-7)
3/-7	3/(-7)
3//-7	3//(-7)
3^-7	3^(-7)

4. Las multiplicaciones necesariamente se tienen que hacer con el operador ‘*’

Error	Corrección
(5)(20)	5*20
(3-4/2)(22-2//1)	(3-4/2)*(22-2//1)

5. Se pueden colocar letras o palabras enteras y van a representar una variable

Expresión	Variable(s)
5+a-6	a
5+alpha-6	alpha
a+b-c^d	a, b, c, d

6. No se pueden colocar letras junto con números en la expresión, tienen que considerarse como operandos independientes y separados en la expresión por algún operador

Error
5alpha
265.beta
gamma.07

7. Para escribir operandos con punto decimal se pueden poner de la siguiente manera:

Operando	**Status**
0.99	Correcto
.99	Correcto

8. Para expresar raíces estas deben de representarse en su versión “exponencial”

Expresión deseada	**Expresión válida**
$\sqrt{2}$	2^(1/2)
$\sqrt[4]{5^{3}}$	5^(3/4)

9. Se pueden asignar valores numéricos o expresiones enteras a las variables que se pongan de entrada en la expresión

33*c-45+s	
Variable	**Ejemplo**
c	210
s	16.07*3//10^3

10. Al colocar variables, en el momento de evaluar la expresión primero se evalúa (si es necesario) lo contenido en la variable. Esto funge como si fuera una expresión contenida en un paréntesis.

Expresión general	
33*c-45+s	
Variable	**Ejemplo**
c	210
s	16.07*3//10^3
Expresión equivalente	
33*(210)-45+(16.07*3//10^3)	

11. Se pueden colocar variables dentro de variables

Expresión: 22-56*a-c^3

a = beta-45/6

beta = 33.2//2+4

c = 34*(alpha-4)

alpha = 37.9-45*d

d = -14

Expresión equivalente: 22-56*((33.2//2+4)-45/6)-(34*((37.9-45*(-14))-4))^3

12. Al colocar dos o más variables con la misma letra o palabra, el programa pregunta por la asignación del valor de la segunda variable y esta puede ser con un valor diferente o con el mismo valor asignado anteriormente. El usuario es el que tiene que especificar.

- **Detalles experimentos**.

Los experimentos se llevaron a cabo en cuatro partes:
Operaciones básicas. En esta primera parte, las expresiones ingresadas eran de complejidad baja, es decir, expresiones sin variables que no pasaban de cifras de tres dígitos, con y sin decimales, tanto positivas como negativas. Las operaciones aritméticas contenidas en dichas expresiones eran básicas: suma, resta, multiplicación y división. El programa las resuelve de manera correcta y respeta la jerarquía de valores. También probamos que regresara un mensaje de error cuando se ingresan los caracteres inválidos que ya fueron mencionados en los aspectos prácticos adicionales.
Expresiones con una variable. Las expresiones ya eran de una mayor complejidad, pues ya contenían exponentes, tanto enteros como fracciones, es decir, raíces. El objetivo principal era que la asignación de valor a esta variable se hiciera de manera correcta, obtuvimos los resultados esperados con números mayores y menores a cero. Aquí probamos la variable en todos los operandos posibles, o sea, siendo sumado, restado, multiplicado, dividido e incluso como exponente. Es importante que la asignación de valores no puede ser otra variable, tiene que ser una cifra forzosamente.
Expresiones con más de una variable. Probamos expresiones complejas con más de una variable, e incluso repitiendolas. Notamos que al repetir una misma variable en una misma expresión debemos asignar el mismo valor las veces que ésta se repita, pues el programa no lo guarda. De todas formas, el programa resuelve las operaciones de manera exitosa, siempre respetando la jerarquía de valores.
Casos especiales. Consideramos el caso en el que el usuario tratase de dividir entre cero. Primero ingresamos una expresión con el cero en el denominador, después consideramos que a una variable en el denominador se le asigne el valor de cero y, por último, el caso en el que se intentara dividir cero entre cero. En todos los casos nos regresó el *ZeroDivisionError*.
También, al ingresar una expresión elevada a un exponente grande, nos marca un *OverflowError* porque el resultado es una cifra de muchos números.

- **Análisis de los resultados**.

En general, es importante mencionar que el programa verifica que las expresiones ingresadas cumplan con los requisitos establecidos, es decir, que los caracteres ingresados sean válidos; que, al usar paréntesis, éstos se encuentren balanceados; que no se pongan dos operandos de manera consecutiva; etcétera. De incumplirse uno de estos criterios, imprime un mensaje de error.

El programa funciona de manera correcta, cada una de las expresiones ingresadas fueron evaluadas a la par con una calculadora. Se llegó al mismo resultado en cada una de las pruebas realizadas.
La asignación de valores a las variables es muy sencilla, y no hay ningún problema, aún ingresando valores negativos, el programa lo resuelve de manera correcta.

El programa marca errores cuando tratamos de hacer operaciones que no tienen respuesta, tal es el caso de la división entre cero; así como también nos marca un error cuando el resultado de una operación arroja un número demasiado grande y esto es bastante interesante. Expresado en notación exponencial o científica, un resultado puede ser expresado por la computadora de este programa hasta cerca del exponente $1\,e^{308}$. Aquí mostramos una pequeña tabla con dos ejemplos:

Expresión	Resultado
10^{308}	$1e + 308$
10^{309}	OverflowError: (34, 'Result too large')
$9.9^{309.606}$	$1.7973392504292843e + 308$
$9.9^{309.607}$	OverflowError: (34, 'Result too large')

Por otro lado, el tope mínimo para un exponente con decimales resulta ser cerca del exponente $1\,e^{-324}$
Aquí mostramos otra tabla con dos ejemplos:

Expresión	Resultado
10^{-323}	$1e - 323$
10^{-324}	0.0
9.9^{-325}	$5e - 324$
9.9^{-326}	0.0

Sin embargo, también notamos que las operaciones en la computadora no son tan precisas cuando se trata de expresiones más complicadas. Al poner 6^6 obtuvimos como resultado el valor 46656, pero al tratar de sacarle la raíz sexta a 46656 (es decir, al probar 46656^(1/6)) nos dio como resultado 5.9999
En algunas operaciones bastante "robustas" el programa llega a resultados que parecen no decir nada. Esto no quiere decir que estén mal, pero tal vez como entrada de datos sí es una operación bastante compleja. A pesar de que el resultado no es entendible, si la entrada de datos se hace por medio de variables o de una expresión general completa, se obtiene el mismo resultado. Veamos un ejemplo:

1. Expresión: a^b-c^(d/e)

a = f-35.24

f = 35/6.32

b = 12.9/g

g = 6.32456

c = 78*(5^(-4))

d = 9.91^(-h)

h = 34.5-34

e = 66-3

Resultado: (1000.4227573874807+125.44544754314192j)

2. Expresión: ((35/6.32)-35.24)^(12.9/(6.32456))-(78*(5^(-4)))^((9.91^(-(34.5-34)))/(66-3))
Resultado: (1000.4227573874807+125.44544754314192j)

- **Conclusiones**.

Fue interesante llevar a cabo esta práctica debido a que existió mucha discusión acerca de los criterios a tomar en cuenta tanto para decidir qué iba a estar dentro de los límites de nuestra calculadora como para diseñar el código implementando y considerando dichos criterios. Si bien fue algo que se realizó de manera separada, los integrantes del equipo siempre estuvieron en comunicación para que el código tuviera la cohesión necesaria y esto hizo que los integrantes conocieran a fondo sobre el funcionamiento del código. Además, es importante conocer otro tipo de código para crear una calculadora (esto lo decimos haciendo referencia a la calculadora programada en la clase de Estructuras de Datos), ya que ofrece una visión más amplia del mundo de la programación.

- **Bibliografía**.

John Sturtz. (s/a). *Strings and Character Data in Python*. Abril, 2021, de Real Python Sitio web: https://realpython.com/python-strings/

Tristan Havelick. (2009). *How do I parse a string to a float or int?* Abril, 2021, de stackoverflow Sitio web: https://stackoverflow.com/questions/379906/how-do-i-parse-a-string-to-a-float-or-int

user3033766. (2013). *OverflowError: (34, 'Resultado demasiado grande')*. Abril, 2021, de it-swarm-es.com Sitio web: https://www.it-swarm-es.com/es/python/overflowerror-34-resultado-demasiado-grande/1042874882/