



Calculadora versión E.D.A.

28/10/2021

Equipo #3

Fecha de entrega : 29/10/2021

Guillermo Arredondo Renero

Luisa Saloma Strassburger

Introducción

El presente trabajo presenta un modo de implementar las estructuras de datos de tipo Pila y Árboles Binarios para la solución de un problema tan cotidiano como lo es el cálculo de operaciones básicas: suma, resta, multiplicación y división. En este caso se diseñó una calculadora que pueda resolver problemas aritméticos de forma independiente del computador de Python, considerando las jerarquías de operaciones habituales, incluyendo o excluyendo paréntesis.

Es importante resaltar que a pesar de haber diseñado una calculadora el semestre pasado en la materia de Estructura de Datos, ésta había presentado numerosas fallas en ciertas evaluaciones de expresiones que por descuido no se habían notado. Dado a que en aquel momento el equipo estaba conformado por los mismos integrantes del presente trabajo (además de otras tres personas), no fue complicado abordar las fallas, errores y soluciones para mejorar los algoritmos de la calculadora. Entre los errores de la calculadora pasada se encontraban los siguientes:

- Utilización de la estructura algorítmica *switch*; la cual, además, no existe en Python.
- Agregar código para soluciones problemas específicos poco a poco en lugar de pensar en una forma general o, en su defecto, limpiar el código para evitar ambigüedades o contradicciones de algorítmica.
- No considerar explícitamente el caso de la división entre cero. Es decir, manejar el caso especial de forma autónoma, distinta de la excepción por parte de Java, en este caso Python.
- Intentar interpretar los posibles errores de sintaxis por parte del usuario.
- Utilizar dos símbolos diferentes para diferenciar los números negativos de la operación de resta.
- Hacer todas las modificaciones y procesos en una sola función, en lugar de usar algunas auxiliares.
- Diferenciar las partes de la operación que se deben mantener unidas; por ejemplo: "22" o "2" y "2" o "-2" y "-", "2".

Todos estos problemas se tomaron en cuenta para mejorar la calidad del programa.

Estrategia de trabajo

Debido a algunas dificultades en el desarrollo de algoritmos por separado, en el presente trabajo se intentó que la gran parte del trabajo se realizará en conjunto, con algunos avances individuales o en subgrupos de vez en cuando. La gran mayoría de los algoritmos se trabajaron en horario de clase y durante reuniones posteriores a la clase.

En el caso de los algoritmos y partes del programa que se dividieron, se establecieron cargas de trabajo balanceadas de acuerdo a los algoritmos que hicieran falta terminar o a las ideas que se le ocurriera a algún integrante para resolver algún problema en cuestión, como algunos señalados en la sección de “Retos y dificultades”. Acorde a lo establecido en el documento de explicación de la práctica, dividimos la solución del problema en tres partes en las cuales se intentó trabajar en conjunto, pero en caso de ser en subgrupos, cada uno aportó una carga diferente de trabajo que en conjunto era revisada y probada para pasar a la próxima solución. Para esto usamos Replit donde cada uno podría agregar parte de lo que avanzó.

Planteamiento del problema

1. ¿Es posible solucionar operaciones aritméticas por medio de árboles y pilas en Python?

A su vez, esta pregunta principal se puede desglosar en otras preguntas acordes a la sección del problema que se desea resolver:

2. ¿Cuáles son los casos de expresiones inválidas posibles?
3. ¿Existe algún caso que pueda obviar de errores por parte del usuario? Por ejemplo, que los paréntesis reflejan multiplicación.
4. ¿Cómo podemos identificar y almacenar los “tokens”?
5. ¿Cómo manejamos los negativos?
6. ¿Qué regresamos en caso de los errores de sintaxis o expresiones inválidas?

Hipótesis

Si ordeno mi expresión aritmética por medio de pilas y la evaluó por medio de un árbol entonces podré obtener un resultado.

Desarrollo de proyecto

Para comenzar con la práctica se realizó una división de problemas en subproblemas que ayudarán a obtener resultados más exactos y claros. Cada subproblema fue pensado y analizado en conjunto, salvo algunos casos especiales, en que fueron divididos entre los integrantes y adaptado y corregido en conjunto para la solución completa del problema. Posteriormente en grupo, se ideó la mejor forma de solucionar dificultades y corregir la eficiencia de código lo mejor posible.

Para comenzar a desarrollar el proyecto teníamos que preparar el área de trabajo; es decir, implementar en python aquellas herramientas que utilizamos en la validación y evaluación, como lo son:

- Implementar y definir la clase Pila (Stack).
- Implementar y definir la clase Árbol Binario.
 - Implementar y definir la clase Nodo (binario).
- Obtención y manejo de "tokens".

Nuestro primer objetivo fue poder identificar los "tokens" de la expresión, después de analizarlo y discutirlo un poco, se pensó en la utilización de una función auxiliar que recibiera la expresión antes de validarla y que la pudiera separar por "tokens", lo cual facilitaría la verificación de la expresión posteriormente. Para la separación de "tokens" se consideró apropiado la conversión de la expresión recibida como cadena en una lista que acumulara los distintos casos de "tokens":

- `getTokens(expresión)`: Función que es invocada para crear "tokens" dada una cadena de texto. La función ordena una lista tomando en cuenta el tipo de carácter que se va leyendo sobre la cadena.
- `tipoDato(caracter)`: Funcion que recibe un caracter como parámetro y es convertido a su valor en código Ascii. Su función es devolver la función que desempeña el caracter en la expresión infija. En este caso, fue necesario un análisis de los valores y organización de la tabla Ascii para darnos cuenta de una forma fácil de generalizar

los distintos tipos de dato: "número", "caracter" y "punto" (decimal). No obstante, posteriormente se agregaron los casos de "parDer", "parIzq" y "masMenos" para facilitar futuras identificaciones de casos especiales.

A partir de la segunda semana ya estábamos comenzando a programar nuestro método de verificación de expresión, dónde buscábamos solamente corroborar que la expresión dada por el usuario fuera válida y tuviera una solución lógica. A partir del desarrollo de esta solución fue necesario crear varias funciones, las cuales se mencionara su nombre y función:

- puntoBien(expresión): Función que revisa un token donde debe existir a lo más un punto por celda de la lista recibida de getTokens().
- valida(expresión): En esta función se verifica que tanto los "tokens" generados sean válidos, como que el orden de los operadores y operandos sea el correcto. Los casos a considerar son:
 - Balance de paréntesis.
 - Repetición de puntos.
 - Operaciones sin dos operandos.
 - Combinaciones de caracteres inválidas (repetición de operadores que no sean de adición o que indiquen signo, "+*", "-*", "+/", "-/", "()").
 - Operación explícita de división entre cero (el caso de división entre cero por efecto de alguna otra operación se consideró durante la evaluación).
 - Expresiones vacías (se consideró como un error).
 - Expresiones sin números (es decir, el caso en que la expresión contenga alguna letra).

La función regresa una tupla en la cual se almacenan en la primera posición una variable de tipo 'bool' de acuerdo a la validez de la expresión y en la segunda posición el mensaje correspondiente ("Math ERROR", "Syntax ERROR" o "").

Una vez verificada que la expresión se puede evaluar y no contiene incoherencias se procede a crear el árbol donde se ordenara la expresión para ser evaluada, de este método ocupamos más funciones para mayor claridad en los procesos, como:

- balanceaSigno(token): Función que recibe un "token" y limpia los signos negativos extras en la expresión, así como quitar los símbolos "+" que sólo reflejaban el signo positivo de algún número. En este punto, es importante ver el funcionamiento de getTokens() en el caso de los signos +/-;
- limpiaTokens(token): Función que maneja un arreglo de "tokens", donde a cada uno le añade un signo o los balancea para su próxima operación;

- `añadeSigno(arr,empieza)`: Función que añade el signo de multiplicación a un arreglo de caracteres dado el caso de `3(5)`, por ejemplo;
- `prioridad(uno,dos)`: función que recibe como parámetros dos caracteres(operandos) donde se pregunta cuál de los dos tiene mayor jerarquía según la jerarquía de operaciones;
- `creaArbol(lista)`: Recibe la lista de `getTokens()`, valida la expresión a través de la función `valida()` y crea un árbol a partir de la expresión 'limpia' de `getTokens()`. Para ello se declaran dos pilas, una de operadores y una que guarda los nodos raíz de los árboles de expresión creados. En este método también se utilizó la función auxiliar `prioridad(uno, dos)` que recibe dos caracteres de tipo operador.

En cuanto a la clase `Árbol`, se decidió que se podía no implementar, puesto que la única funcionalidad que tendría sería el recorrido en "postorden", el cual podría implementarse de forma autónoma al guardar la dirección del nodo "raíz".

Por último, creamos los métodos de `evalua` que esta dividido en dos partes, la primera donde se ordenan todos los algoritmos explicados de manera que en una función se condensan todos los sub programas:

- `evalua(expresion)`: recibe la expresion y crea tanto los "tokens" como el árbol en función de los tokens para que devuelva el resultado de `evaluaRec()` con el árbol ya verificado y creado.
- `evaluaRec(nodo)`: verifica que el nodo recibido no sea nulo, luego realiza un recorrido en postorden y conforme va sacando izquierdos y derechos se pregunta, si el dato que obtengo es un signo, evaluó la parte izq y la derecha, si es un operando regresa el dato, pues estos siempre se encuentran en las hojas de los árboles; si es una división y la parte der es un 0, entonces la expresión aritmética es incorrecta y regresa la cadena "Math ERROR".

Para poder evaluar nuestra calculadora diseñamos un método *main* donde escribimos 35 expresiones aritméticas en las cuales probamos las diferentes partes del código y de esta manera conocer aquellos puntos faltantes donde la expresión no debería funcionar o no funciona. En el caso de que se descubriera algún error, se buscaba como solucionarlo y se comprobaba con el uso de una calculadora CASIO, además de usar excel como apoyo y nosotros mismos evaluar las operaciones.

Especificaciones

Realizamos una calculadora que por medio de árboles, pilas y tokens, donde se verifica la validez de una expresión aritmética y se regrese un valor correcto. Es importante resaltar algunos detalles:

1. Se utilizó una lista para poder manejar adecuadamente los “tokens” de acuerdo a algunas validaciones
2. Se consideraron sintácticamente correctas las siguientes expresiones: “++++5”, “+-+88.9”, “-----1”, “*-”, “/-”, “(5+8)(8-4)”, “+5(6-4)”, “-5(8+6)” y sus respectivas derivaciones.
3. Como se dijo anteriormente, no se implementa la clase Árbol, sino que se crea manualmente insertando a la izquierda o a la derecha según corresponda.
4. El algoritmo para la segunda fase, la creación del árbol de expresión y su respectiva evaluación, se separaron en dos programas para facilitar la comprensión, sobre todo por el hecho de que el algoritmo de evaluación se estableció de forma recursiva. Sin embargo, estos métodos funcionan perfectamente llamándose uno al otro, sin la necesidad de la llamada al método evalua(). El equipo pensó que esta organización presentaría mejor la información de código.
5. Se intentó generar expresiones aleatorias por medio de un algoritmo, que no resultó.
6. Se utilizaron los valores de la tabla ASCII para identificar casos a través de una función auxiliar.
7. Se implementó un procesamiento de expresiones que dieran como resultados parciales una división entre cero.
8. Se utilizaron tuplas para transmitir dos datos de una función a otra.
9. Se creó un atributo “Node.papa” para la clase Node, pero en realidad no fue necesario utilizarla.

Conocimiento adicional

Uno de los principales conceptos teóricos que fue necesario recordar fueron todas las especificaciones necesarias para la Programación Orientada a Objetos en Python. Con apoyo de los apuntes de clase y las explicaciones del profesor, se implementaron ambas clases Pila y Node para ser utilizadas durante los algoritmos de evaluación.

A su vez, para esta práctica fue necesario saber un poco más sobre el código ASCII, ya que de esta forma podríamos definir intervalos de operadores, números y aquellas expresiones que no se podrían admitir de manera numérica; es decir, definir un intervalo para números,

otro para operadores y caracteres especiales. Esto resultó sencillo, pues los caracteres de operaciones (paréntesis, suma, resta, multiplicación, división y punto) se encuentran de forma continua en la tabla ASCII asumiendo valores del 40 al 47, exceptuando el 44 que es el caso de la coma, y los números se encuentran del 48 al 57. Esto puede verse en la tabla a continuación:

Caracteres ASCII de control					Caracteres ASCII imprimibles					ASCII extendido (Página de código 437)									
00	NULL	(carácter nulo)	32	espacio	64	@	96	`		128	Ç	160	á	192	Ł	224	Ó		
01	SOH	(inicio encabezado)	33	!	65	A	97	a		129	ù	161	í	193	ł	225	ß		
02	STX	(inicio texto)	34	"	66	B	98	b		130	é	162	ó	194	Ł	226	Ô		
03	ETX	(fin de texto)	35	#	67	C	99	c		131	â	163	ú	195	ł	227	Ò		
04	EOT	(fin transmisión)	36	\$	68	D	100	d		132	ä	164	ñ	196	—	228	ö		
05	ENQ	(consulta)	37	%	69	E	101	e		133	à	165	Ñ	197	†	229	Õ		
06	ACK	(reconocimiento)	38	&	70	F	102	f		134	á	166	ª	198	‡	230	μ		
07	BEL	(timbre)	39	'	71	G	103	g		135	ç	167	º	199	Ä	231	þ		
08	BS	(retroceso)	40	(72	H	104	h		136	ê	168	¿	200	Ł	232	ƒ		
09	HT	(tab horizontal)	41)	73	I	105	i		137	ë	169	®	201	ł	233	ù		
10	LF	(nueva línea)	42	*	74	J	106	j		138	è	170	¬	202	Ł	234	û		
11	VT	(tab vertical)	43	+	75	K	107	k		139	ï	171	½	203	ł	235	ü		
12	FF	(nueva página)	44	,	76	L	108	l		140	î	172	¾	204	ł	236	ý		
13	CR	(retorno de carro)	45	-	77	M	109	m		141	ï	173	ı	205	≡	237	ÿ		
14	SO	(desplaza afuera)	46	.	78	N	110	n		142	Ā	174	«	206	≡	238	—		
15	SI	(desplaza adentro)	47	/	79	O	111	o		143	Ă	175	»	207	□	239	˙		
16	DLE	(esc.vínculo datos)	48	0	80	P	112	p		144	É	176	≈	208	ð	240	≡		
17	DC1	(control disp. 1)	49	1	81	Q	113	q		145	æ	177	≡	209	Ð	241	±		
18	DC2	(control disp. 2)	50	2	82	R	114	r		146	Æ	178	≡	210	È	242	—		
19	DC3	(control disp. 3)	51	3	83	S	115	s		147	ô	179	ı	211	Ê	243	¾		
20	DC4	(control disp. 4)	52	4	84	T	116	t		148	ö	180	ı	212	Ë	244	ŋ		
21	NAK	(conf. negativa)	53	5	85	U	117	u		149	ò	181	Ā	213	Ĭ	245	§		
22	SYN	(inactividad sinc)	54	6	86	V	118	v		150	û	182	Ă	214	İ	246	÷		
23	ETB	(fin bloque trans)	55	7	87	W	119	w		151	ù	183	Ä	215	İ	247	ˆ		
24	CAN	(cancelar)	56	8	88	X	120	x		152	ÿ	184	©	216	Ĵ	248	˚		
25	EM	(fin del medio)	57	9	89	Y	121	y		153	Ö	185	®	217	Ĵ	249	˚		
26	SUB	(sustitución)	58	:	90	Z	122	z		154	Ü	186	≡	218	Œ	250	˚		
27	ESC	(escape)	59	;	91	[123	{		155	ø	187	Œ	219	■	251	˚		
28	FS	(sep. archivos)	60	<	92	\	124			156	£	188	Œ	220	■	252	˚		
29	GS	(sep. grupos)	61	=	93]	125	}		157	Ø	189	¢	221	ı	253	˚		
30	RS	(sep. registros)	62	>	94	^	126	~		158	×	190	¥	222	ı	254	■		
31	US	(sep. unidades)	63	?	95	_				159	f	191	Œ	223	■	255	nbsp		
127	DEL	(suprimir)																	

González, S. (2016). *Tabla ASCII*. Obtenida de: <https://blogsilviagonzalez.blogspot.com/2016/04/febrero-232016.html>

Retos y dificultades

Una de las primeras dificultades que nos encontramos fue decidir si implementar solo nodos o una clase árbol para la solución. Dado que de la estructura del árbol solo necesitamos el recorrido de postorden, solo implementamos un nodo con dos nodos, "izq" y "der", con la funcionalidad del recorrido en postorden a partir de un cierto nodo raíz.

Pequeños retos y dificultades representó obtener los casos en que la calculadora no debería funcionar, tales como la división entre cero, operadores juntos (sin tomar en cuenta los casos de los signos de adición), el doble punto dentro de un operando, paréntesis balanceados, paréntesis entre operadores [sin tomar en cuenta los casos de ")-", ")+", "+(", "-("].

Otra dificultad se presentó en el caso de la generación del método postOrden(). Como tal, este método era un mecanismo auxiliar para ejemplificar y observar la correcta estructura del árbol de expresión resultante, pues no es utilizado para la función de evaluación. Sin embargo, el método funcionaba erróneamente. Después de un análisis profundo de los algoritmos y asesoría por parte del profesor, se descubrió que el error no estaba en el método recursivo de postOrden(), sino en la forma en que se insertaban los datos. Afortunadamente, este error se observó a tiempo y ayudó en modificar el algoritmo, pues de haber mantenido el error, varias expresiones de mayor complejidad habrían concluido con un resultado incorrecto.

Dado que nuestro método de postorden nos regresa un arreglo de signos y operandos, entonces, tuvimos conflicto en decidir si evaluar con postorden y de manera recursiva o utilizar el arreglo directamente. Al final evaluamos recursivamente.

Otro reto a considerar es solucionar que el usuario al momento de escribir dos expresiones en forma de multiplicación con paréntesis, pero sin signo de multiplicación, la calculadora añadiera el signo de multiplicación y de esta manera poder evaluar en el árbol aquella expresión.

Además, consideramos cuando una expresión en el denominador su resultado es cero o si el numerador es cero, de esta manera nos encargamos de manejar el error y no dejar a Python generar una excepción. En suma, expresiones que inician con un signo positivo o negativo son tomadas en cuenta como válidas; mientras que, expresiones inconclusas donde solo existe un operador y un operando son consideradas erróneas.

Un error que nos costó trabajo identificar, ocasionado por la forma en que está diseñado nuestro método de evaluación, es que la división entre 0 devuelve una cadena de texto

“Math ERROR”; por lo tanto, cuando el árbol encuentra un operando y de algún lado tiene el mensaje de error, no sabe que hacer, ya que por un lado contiene un número y por el otro una cadena de texto lo cual despliega un error en la ejecución de la operación recursiva inmediata anterior. Este error se solucionó añadiendo una condición extra que preguntara si alguna de las dos partes a evaluar era de tipo “str”.

Para el diseño de las pruebas intentamos crear una función donde nos regresara una expresión aritmética aleatoria, pero la complejidad del algoritmo nos guió a desistir y mejor establecer la expresión por cuenta propia. Esto resultó ser de ayuda, pues nos permitió analizar el resultado por nuestra cuenta.

Resultados y Análisis de Resultados

Encontramos que nuestra calculadora soluciona correctamente las expresiones válidas dadas y regresa la solución de forma correcta; mientras que, aquellas expresiones que tanto son invalidadas por la forma en que fueron escritas, manejan correctamente los errores.

Operación	Resultado esperado	Resultado calculadora
$(6+2)(5-3)$	16	16
$(5+3/2)(7+2)$	58.5	58.5
$5+3-10$	-2	-2
$8/2*(2+2)$	16	16
$6/2*(1+2)$	9	9
$9-3/1/3+1$	9	9
$9*-5+++++89/-75+(38)-2*5$	-18.1866	-18.18666667
$(89)-5$	84	84


$(3.14+((4+8)/3))(15.1-3(3/2+1))$	54.264	54.264
$5*6-((-4/3-1)45-56/34)+1$	137.647059	137.6470588
1	1	1
()	SE	Syntax ERROR
(2)	2	2
(SE	Syntax ERROR
$((5+8-9)(84)(-35+54/9)$	SE	Syntax ERROR
$38-9/15^*+85+^*17$	SE	Syntax ERROR
$38-9/15^*+85*17$	-829	-829
$5*8/0$	Math ERROR	Math ERROR
$84-8*54^{**}5$	SE	Syntax ERROR
0/5	0	0
$38-9/(15-15)^*+85*17$	MATH ERROR	Math ERROR
$38-9/15^*+85+^*17)$	SE	Syntax ERROR
$105-84*.2+(87/11)/5+(-8*9/(5-(-3)))$	80.781818	80.78181818
$35-(88+55(17-(15*32-(88/79))/9+17*19/66))$	1669.359353	1669.359353
$35-(88+55(17-(15.9*32-(88.5/79)))/9+17*19.892/66)$	2940.474756	2940.474756

0/0	Math ERROR	Math ERROR
$(-15+15)/(15-15)$	Math ERROR	Math ERROR
0//0	Syntax Error	Syntax ERROR
$+ 5 / \text{-----} 1$	5	-5
a+b	SE	Syntax ERROR
VACÍO	Syntax Error	Syntax ERROR
5+b	res esperado	SE
VACÍO	res esperado	SE
5+	res esperado	SE
/	res esperado	SE

En cuanto a resultados parciales, a continuación se agregan algunos ejemplos del funcionamiento de los métodos `getTokens()` y `limpaTokens()` y el `postOrden()` del árbol generado:

Expresión	"Tokens"	postOrden()
"----5(4-3.5)-8"	['5', '*', '(', '4', '-', '3.5', ')', '-', '8']	['5', '4', '3.5', '-', '*', '8', '-']
"(6+2)(5-3)"	['(', '6', '+', '2', ')', '*', '(', '5', '-', '3', ')']	['6', '2', '+', '5', '3', '-', '*']
$5*6-((-4/3-1)45-56/34)+1$	['5', '*', '6', '-', '(', '(', '-4', '/', '3', '-', '1', ')', '*', '45', '-', '56', '/', '34', ')', '+', '1']	['5', '6', '*', '-4', '3', '/', '1', '-', '45', '*', '56', '34', '/', '-', '-', '1', '+']

Puede observarse que los métodos funcionan a la perfección en cualquiera de los casos evaluados. Es importante aclarar que muchas de las expresiones fueron pedidas a terceros, buscadas en internet o redes sociales como retos de operaciones de jerarquía; por lo cual, cuentan con diversidad y cierto grado de aleatoriedad que ejemplifican con mayor detenimiento su eficacia en la resolución de operaciones aritméticas básicas. A su vez, se probaron varios de los casos en contraste con lo que una calculadora CASIO



profesional haría, qué casos sí evalúa, cuáles considera correctos y la expresión se ingresó exactamente igual que en el algoritmo creado. A través de las pruebas de varios casos particulares se observaron ciertos errores que ayudaron a poder mejorar el algoritmo. Todos los resultados fueron satisfactorios, finalmente.

Conclusiones

La calculadora programada en este proyecto resultó ser competente en la resolución de problemas aritméticos básicos e incluso podría asegurarse que lo ejecuta con mayor facilidad con respecto a la calculadora programada el semestre pasado en la materia de Estructura de Datos. La implementación de la clase Árbol en la solución del problema ayuda en la determinación de casos y en la conversión de expresiones infijas a post fijas de forma excepcional. Además, el uso de algoritmos recursivos en lugar de iterativos tiene una ventaja ineludible.

Por último, el equipo quiere expresar sus conclusiones respecto a que Python resultó ser de mayor utilidad en la consideración de los “tokens”, al menos para los integrantes de este equipo, conseguir esto en java en la materia de Estructura de Datos presentó un reto muy complejo de resolver. Algo que en Python no solo fue más sencillo, sino que además permitió solucionar problemas futuros, como el caso del signo negativo, el cual era posible añadirse al mismo “token” del número inmediato siguiente.

Consideramos que el presente proyecto tiene grandes aplicaciones en la resolución de problemas, estamos orgullosos del trabajo realizado y consideramos que nos ha permitido solidificar las bases de programación en Python en todas sus ramas y el aprovechamiento de su versatilidad.

Bibliografía

González, S. (2016). Tabla ASCII. Obtenida de:

<https://blogsilviagonzalez.blogspot.com/2016/04/febrero-232016.html>

Anónimo (sin fecha). El código ASCII. Obtenida de: <https://elcodigoascii.com.ar/>

Creative Commons Atribución-NoComercial 3.0 Unported (2013). Cómo obtener el código ASCII de un carácter y viceversa. Obtenida de:

<https://micro.recursopython.com/recursos/como-obtener-el-codigo-ascii-de-un-caracter.html>