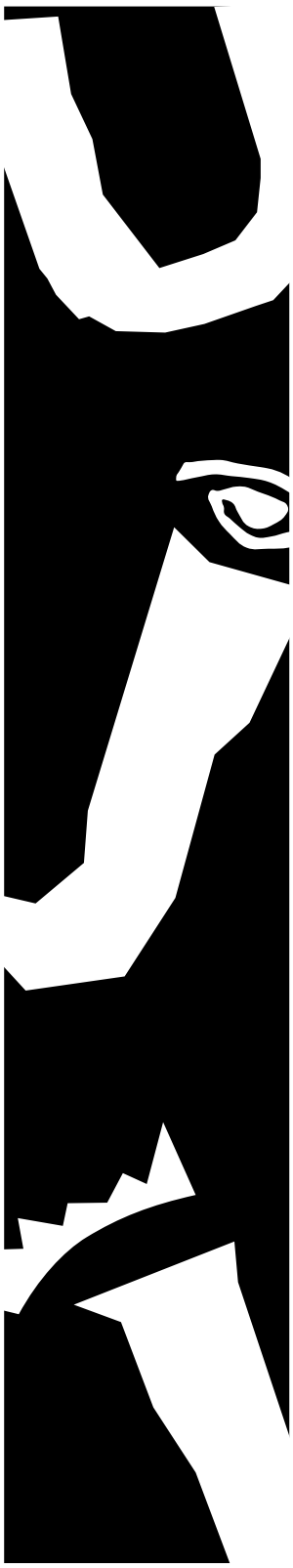


UNIVERSITAT JAUME·I

4º Ingeniería Informática

II26 Procesadores de lenguaje

Práctica 1: microcalc, una calculadora elemental

Índice

Introducción

En clase de teoría se ha estudiado la estructura general de los procesadores de lenguaje, viendo que se pueden descomponer en una fase de análisis, cuyo objetivo es acabar obteniendo un AST que representa la entrada, y una fase posterior que, en el caso de los intérpretes, consiste en su evaluación mediante un recorrido recursivo del AST. A su vez, la fase de análisis consta de tres etapas: análisis léxico, análisis sintáctico y análisis semántico.

El objetivo de esta práctica es que seas capaz de aplicar esto a un caso real, aunque de momento relativamente sencillo. Para ello, te proponemos implementar una calculadora que evalúe expresiones con enteros y cadenas. A lo largo del curso veremos cómo crear intérpretes con más posibilidades en cuanto a las expresiones que se pueden manejar. Sin embargo, los principios de organización serán muy similares a los que seguirás aquí.

Este documento está dividido en dos partes. En la primera, se describe el problema a resolver a lo largo de toda esta práctica, y se detalla el comportamiento de la calculadora y las características del lenguaje de entrada. La segunda parte es una guía de desarrollo, con orientaciones sobre el diseño y la implementación de la calculadora. En ella te propondremos seguir un ciclo de desarrollo incremental: empezaremos por una calculadora básica de funcionalidad limitada, pero con un diseño orientado desde el principio a facilitar la posterior incorporación de extensiones en versiones sucesivas.

Te recomendamos que te asegures de que la calculadora básica y cada ampliación funcionan correctamente antes de pasar a la siguiente. Para ello, deberás diseñar pruebas, tanto de entradas correctas como erróneas. Además, te resultará útil implementar las opciones que se detallan en la sección 1 para depurar errores.

El desarrollo lo realizarás en Linux, utilizando Python como lenguaje de programación. Por lo demás, puedes elegir libremente el editor o entorno de programación que más te guste.

Parte I

Especificación

1. Invocación y comportamiento de la calculadora

La calculadora debe poder ejecutarse interactivamente escribiendo en un terminal la orden:

`./microcalc [`*`opción`*`]`

Su labor es analizar y ejecutar cada línea que le llegue por la entrada estándar y, si no hay errores, mostrar el resultado en la salida estándar. En caso de que la línea leída contenga algún error, se mostrará en la salida de error un mensaje en ASCII (sin acentos) que indique el número de línea (contando desde uno) y el tipo del primer error encontrado ("`Linea` *`n`*`: Error lexico.`", "`Linea` *`n`*`: Error sintactico.`", "`Linea` *`n`*`: Error semantico.`" o "`Linea` *`n`*`: Error de ejecucion.`") y se analizará la línea siguiente. Los posibles efectos secundarios de la línea no sucederían (esto es, no se escribiría nada por la salida estándar).

Con la opción `-l` debe mostrar la secuencia de componentes léxicos, con la opción `-a` debe mostrar el árbol de derivación, y con la opción `-s` debe mostrar el AST, todo ello por la salida estándar[1]. Dichas opciones son excluyentes e inhiben la salida de resultados. El comportamiento de estas opciones ante líneas de entrada erróneas está indefinido.

2. Estructura de la entrada

La entrada de `microcalc` es una secuencia de líneas, cada una de las cuales tiene una expresión que se evalúa y cuyo resultado se muestra por la salida estándar seguido de un salto de línea. Los resultados, tanto enteros como cadenas, se muestran en el mismo formato que la sentencia `print` de Python.

Algunas expresiones en `microcalc` y sus correspondientes resultados son:

Expresión	Resultado
`2+3*5`	`17`
`7*(4+2)`	`42`
`|"a\tb"|`	`3`
`|3*"ab"|-2`	`4`
`3*"a\"b"`	`a"ba"ba"b`
`||"ab"|*"cde"|*|"fg"+"hi"|`	`24`

Las líneas vacías (formadas sólo por el carácter salto de línea, o por componentes que se omiten y el salto de línea) se consideran erróneas. También son incorrectas las líneas que no finalizan con salto de línea (esto podría suceder en la última línea de un fichero de entrada).

3. Algunos elementos léxicos

3.1. Literales

En `microcalc` hay dos tipos de literales:

Literales enteros: el carácter `0` o cualquier secuencia de dígitos que comience por uno distinto del `0`.

Literales de cadena: secuencias de caracteres encerradas entre comillas dobles. Se admiten cuatro secuencias de escape, `\\`, `\"`, `\n` y `\t` que representarán los caracteres barra invertida, comillas, fin de línea y tabulador. Estos caracteres no pueden aparecer de otra forma entre las comillas que delimitan la cadena.

[1] Para poder ver los árboles, puedes escribirlos en el formato de la herramienta VerArbol.

3.2. Componentes léxicos que se omiten

Los componentes léxicos pueden separarse por cualquier secuencia de espacios en blanco y tabuladores, que son omitidos.

4. Tipos de datos

El lenguaje dispone de dos tipos de datos: entero y cadena.

5. Expresiones

En `microcalc` son expresiones válidas los literales (enteros y cadenas). Además, también son válidas aquellas expresiones que pueden construirse a partir de otras utilizando correctamente los operadores ofrecidos por el lenguaje. Finalmente, cualquier expresión válida puede encerrarse entre paréntesis para formar así una nueva expresión válida.

El lenguaje dispone de los siguientes operadores:

Operador suma: `+`. Es binario, infijo y asociativo por la izquierda. Ambos operandos deben ser del mismo tipo, que coincide con el del resultado. En el caso de las cadenas, se interpreta como concatenación de sus operandos.

Operador resta: `-`. Es binario, infijo y asociativo por la izquierda. Ambos operandos deben ser de tipo entero, igual que el resultado.

Operador producto: `*`. Es binario, infijo y asociativo por la izquierda. Si ambos operandos son de tipo entero, el resultado es de este tipo. Si uno de los operandos es de tipo cadena y el otro entero, el resultado es una cadena formada concatenando la cadena dada el número de veces indicado por el entero. Si el entero es negativo o cero, el resultado es la cadena vacía.

Operador división: `/`. Es binario, infijo y asociativo por la izquierda. Ambos operandos deben ser de tipo entero, igual que el resultado. Si el divisor es cero, se produce un error de ejecución.

Operador cambio de signo: `-`. Es unario y prefijo. El operando es de tipo entero, igual que el resultado.

Operador identidad: `+`. Es unario y prefijo. El resultado es el valor del operando, que debe ser entero.

Operador barra: `|·|`. Es unario. El operando debe ser una expresión de tipo entero o cadena encerrada entre las dos barras verticales. Si el operando es de tipo entero, el resultado es su valor absoluto; si es de tipo cadena, su longitud.

El orden de prioridad de los operadores es el siguiente, de menor a mayor:

1. Suma y resta.
2. Producto y división.
3. Cambio de signo e identidad.

En las expresiones se pueden emplear paréntesis para agrupar subexpresiones y alterar el orden de evaluación, de la forma habitual. Por otro lado, el operador barra no necesita un nivel de prioridad explícito puesto que se aplica sobre la expresión que encierra.

Parte II

Guía de desarrollo

6. La calculadora básica

Comenzaremos por una calculadora muy básica. Admitirá únicamente enteros de un dígito, que podrá sumar o multiplicar. No tendrá paréntesis pero sí seguirá las reglas habituales de precedencia y asociatividad: el producto será más prioritario que la suma y ambas operaciones serán asociativas por la izquierda.

6.1. Estructura de la calculadora

El fichero principal será `microcalc` y se encargará de leer las líneas de la entrada estándar, construir los analizadores léxico y sintáctico y utilizar el AST para calcular el resultado. Además se ocupará del tratamiento de errores.

Para el analizador léxico, utilizaremos la clase `Lexico`, que residirá en `lexico.py` y que será capaz de dividir la línea de entrada en componentes. Estos componentes estarán definidos en `componentes.py`.

Como se comenta en el tema de teoría sobre estructura de los compiladores e intérpretes, será el analizador sintáctico el que "lleve la batuta". Para ello, tendremos en `sintactico.py` la definición de una clase, `Sintactico`, que hará las llamadas oportunas al analizador léxico y construirá el AST correspondiente a la línea leída. Para la construcción del AST, contará con las clases que definiremos en `AST.py`.

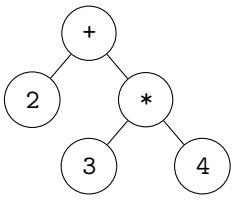
Finalmente, tendremos dos ficheros auxiliares: `errores.py` y `tipos.py`. El fichero `errores.py` simplemente contendrá cuatro clases derivadas de `Exception`: `ErrorLexico`, `ErrorSintactico`, `ErrorSemantico` y `ErrorEjecucion`. De momento, deja las clases vacías (el cuerpo sólo tiene `pass`).

El módulo `tipos.py` definirá dos variables: `Entero` y `Cadena`. Para `microcalc`, nos bastará con que contengan sendas cadenas con el nombre del tipo. En casos más complicados, interesará que sean objetos con información adicional.

Puede que eches a faltar un módulo para el análisis semántico. En realidad, es la fase más dispersa puesto que está mezclada con el análisis sintáctico y con los AST. De todas formas, en una calculadora tan sencilla como está, el semántico tiene un papel muy reducido.

6.2. Representación de las expresiones mediante ASTs

Como sabes, representaremos las expresiones internamente mediante Árboles de Sintaxis Abstracta, a los que llamamos AST. Por ejemplo, la expresión $2 + 3 * 4$ se podría representar internamente mediante el árbol:



Empieza creando un módulo, `AST.py`, que contenga tres clases para representar los tres posibles nodos que encontraremos en los árboles:

- `NodoSuma`: representa una suma. Tiene dos hijos, `izdo` y `dcho`.
- `NodoProducto`: representa un producto. Tiene dos hijos, `izdo` y `dcho`.
- `NodoEntero`: representa un literal entero. Tiene un atributo: `valor`.

El constructor para los nodos que tienen hijos recibe dos parámetros, uno por hijo. En el caso del `NodoEntero`, el constructor recibe directamente el valor. Con esta convención, si queremos representar la expresión anterior, podemos escribir:

```
NodoSuma(NodoEntero(2), NodoProducto(NodoEntero(3), NodoEntero(4)))
```

Para poder evaluar las expresiones, añade a cada nodo el método `evalua`. Este método no tiene parámetros y devuelve el valor de la expresión correspondiente. En el caso de los enteros, el valor es el que se ha almacenado. Para las expresiones, habrá que evaluar el hijo izquierdo, después el derecho y, finalmente, sumar o multiplicar los valores para devolver el resultado.

Si quieres comprobar que las expresiones se representan correctamente en memoria, puedes añadir el método `__str__` a los nodos. Este método devuelve una representación "agradable" del objeto. En nuestro caso, debes hacer lo siguiente: el método `__str__` de los enteros devolverá el valor convertido en cadena. En el caso de las sumas y productos, puede devolver una cadena formada por: un paréntesis abierto, el hijo izquierdo, el símbolo de la operación, el hijo derecho y un paréntesis cerrado. Esto te servirá para implementar la opción `-s`.

6.3. Especificación léxica

Ya tenemos una manera de representar las expresiones internamente y de trabajar con ellas. Ahora tenemos que lograr traducir cadenas de caracteres (la entrada del usuario) a esas representaciones.

Nuestro analizador léxico tendrá inicialmente las siguientes categorías:

Categoría	Expresión regular	Atributos	Acciones
suma	`\+`	—	emitir
producto	`\*`	—	emitir
entero	[0–9]	`valor`	calcular `valor` emitir
blanco	[␣\t]	—	omitir
nl	`\n`	—	emitir
eof	$^{\circ}_{f}$	—	emitir

Por un lado, están los operadores; por otro, los literales enteros (de un dígito). También tendremos que tratar los blancos y los fines de línea. Los primeros, para limpiarlos; los segundos, para indicar que se termina la expresión. Finalmente, debemos indicar de alguna forma al analizador sintáctico que se ha terminado la entrada. Para esto utilizamos la categoría **eof**. Puede parecer redundante tener un fin de línea y un fin de fichero. Piensa, sin embargo, que puede darse el caso de que una línea no esté terminada por un carácter fin de línea, por ejemplo si, en lugar de introducirla por el teclado, lo hacemos redireccionando un fichero. En este caso, ¿qué categoría debería devolver el analizador léxico? Por otro lado, es bueno tener previsto qué devolver si se pide un nuevo componente tras el **nl**.

6.4. Implementación del analizador léxico

Vamos a descomponer el análizador léxico en dos partes: la definición de los componentes léxicos y el analizador en sí. Crea un módulo, `componentes.py`, con una clase por cada categoría de las que no se omiten. Cada clase Python tendrá un atributo `cat`, que nos indique a qué categoría pertenece el componente léxico[2]. Además, la clase **entero** tendrá el atributo `valor` que habrá que calcular en el constructor a partir del lexema.

Para poder comprobar el funcionamiento de tu analizador, querrás escribir cómodamente los componentes léxicos. Para ello, puedes añadir a cada objeto el método `__str__`. Puedes usar la convención de escribir un componente mediante el nombre de su categoría y, en el caso de los enteros, mediante el nombre seguido de la cadena "`valor:`" y el valor entre paréntesis. De este modo, con

```
print componentes.Entero('7')
```

[2]Observa que este atributo no está en la tabla anterior; en realidad, es algo que nos facilitará la implementación, pero no pertenece a los atributos que consideramos como parte de la especificación léxica.

obtendrías el mensaje "`entero (valor: 7)`" por la salida estándar.

Ahora vas a definir una clase para representar el analizador léxico. Para ello, crea en el fichero `lexico.py` la clase `Lexico`. El comportamiento de los objetos de esta clase será el siguiente. Al crearlos, les pasaremos la cadena que se va a analizar, presumiblemente leída por teclado. Posteriormente, iremos llamando a un método, `siguiente`, que irá devolviendo en cada llamada un nuevo componente léxico. Por ejemplo, la secuencia

```
lexico= Lexico('2 + 3*4\n')
while True:
  componente= lexico.siguiente()
  print componente
  if componente.cat== 'eof':
    break
```

deberá escribir por pantalla algo similar a

```
entero (valor: 2)
suma
entero (valor: 3)
producto
entero (valor: 4)
nl
eof
```

Fíjate en que los blancos han "desaparecido", ya que la especificación léxica indica que se deben omitir.

Escribe el constructor de la clase `Lexico`. Como parámetro, recibe una cadena que representa la línea que contiene la expresión. Además de copiar la cadena, inicializa a cero un atributo (por ejemplo `pos`) que indicará en qué posición de la cadena esperamos encontrar el siguiente componente.

Crea ahora un método, `siguiente`, sin parámetros y que al ser llamado devuelva el siguiente componente que se encuentre en la cadena y que no haya que omitir. Puedes implementar este método de dos formas. Puedes diseñar e implementar una MDD, o, dado que este caso es especialmente sencillo, puedes hacer que `siguiente` se comporte así:

- Si `pos` es mayor o igual que la longitud de la cadena, devuelve un componente **eof**.
- Si lo que hay en la posición `pos` de la cadena es un blanco o un tabulador, avanza `pos` y busca otro componente.
- Si se encuentra un dígito, devuelve un componente **entero** inicializado con el carácter correspondiente.
- Si se encuentra un más (`+`), un por (`*`) o un carácter fin de línea, devuelve el componente correspondiente.
- Antes de cada una de las devoluciones, excepto de `eof`, incrementa la posición de `pos` en uno.
- Si se encuentra un error (un carácter que no esté en la lista anterior), eleva una excepción de tipo `ErrorLexico`.

6.5. Análisis sintáctico y construcción del AST

Nuestro objetivo ahora es construir, a partir de lo que nos vaya devolviendo el analizador léxico, ASTs que se correspondan con las expresiones que se introduzcan por el teclado. Las expresiones

que podemos encontrar se construyen a partir de la siguiente gramática[3]:

$$\begin{aligned}
\langle \text{Línea} \rangle &\rightarrow \langle \text{Expresión} \rangle \ \textbf{nl} \\
\langle \text{Expresión} \rangle &\rightarrow \langle \text{Término} \rangle (\textbf{suma} \ \langle \text{Término} \rangle)^* \\
\langle \text{Término} \rangle &\rightarrow \langle \text{Factor} \rangle (\textbf{producto} \ \langle \text{Factor} \rangle)^* \\
\langle \text{Factor} \rangle &\rightarrow \textbf{entero}
\end{aligned}$$

¿Cómo se lee esto? La primera regla dice que una línea en la entrada se compone de una expresión y de un fin de línea; la segunda nos dice que, para construir una expresión, tendremos que tener un término seguido de cero o más repeticiones de un operador suma y un término; la tercera indica la forma de los términos y la última dice que un factor es simplemente un literal entero.

Ahora implementarás un analizador para esta gramática, que construirá el correspondiente AST. Para ello, crea el fichero `sintactico.py` con la clase `Sintactico`, cuyo constructor recibe un analizador léxico que almacena en el atributo `lexico`. Además, llama a `siguiente` del analizador léxico y almacena el resultado en el atributo `componente`.

La parte principal de `Sintactico` va a ser un conjunto de métodos que se corresponderán con los no terminales de la gramática. Estos métodos serán `analizaLinea`, `analizaExpresion`, `analizaTermino` y `analizaFactor`. No tendrán parámetros y su ejecución supondrá analizar la entrada desde el punto donde se obtuvo el último valor de `componente` hasta que se haya encontrado una línea, una expresión, un término o un factor, respectivamente, y devolver el árbol correspondiente.

Así, `analizaLinea` llama a `analizaExpresion` y después comprueba que la categoría de `componente` es igual a `'nl'`. Si todo ha ido bien, se avanza al siguiente componente en el analizador léxico (así, el análisis de la línea consume su salto final) y se devuelve el árbol que `analizaLinea` ha recibido de `analizaExpresion`. En caso contrario, se eleva una excepción de tipo `ErrorSintactico`.

Más interesante es `analizaExpresion`. Hace lo siguiente:

- Primero busca un término mediante `analizaTermino`. El árbol devuelto lo almacena en la variable `arbol`.
- Después, mientras encuentre un operador suma, avanza en el léxico, y busca otro término. Con el árbol correspondiente a este término como hijo derecho y el que tenemos en `arbol` como izquierdo, crea otro árbol que guarda en `arbol`.
- Finalmente (cuando no encuentra más operadores suma), devuelve `arbol`.

En código:

```
def analizaExpresion(self):
  arbol= self.analizaTermino()
  while self.componente.cat== 'suma':
    self.componente= self.lexico.siguiente()
    dcho= self.analizaTermino()
    arbol= AST.NodoSuma(arbol, dcho)
  return arbol
```

El método `analizaTermino` es análogo.

Finalmente, `analizaFactor` comprueba que en `componente` hay un entero. Si no, eleva una excepción de tipo `ErrorSintactico`. Si efectivamente hay un entero, crea un árbol con `NodoEntero` utilizando el valor que tiene el componente y, tras avanzar en el léxico, devuelve ese árbol.

[3]Esta gramática tiene partes derechas regulares, es una generalización de las que conoces por TALF y que veremos en el tema de análisis sintáctico.

6.6. Programa principal

Finalmente, vamos a escribir el programa principal. Llama al fichero `microcalc`. Tras importar los módulos necesarios, escribe un bucle que haga lo siguiente:

- Intenta leer una línea de `sys.stdin` mediante `readline`.
- Si no hay ninguna línea más (`readline` ha devuelto la cadena vacía), sale del bucle.
- En caso contrario, crea un objeto de la clase `Lexico` y otro de la clase `Sintactico`.
- Llama a `analizaLinea` del objeto de la clase `Sintactico` dentro de una sentencia `try` que capture los errores léxicos y sintácticos. En caso de que se produzca alguno de ellos, muestra, por `stderr`, el mensaje "Error léxico" o "Error sintáctico", según corresponda.
- Si no ha habido errores, escribe, por `stdout`, el resultado de evaluar la expresión.

Determina si es posible que se produzca un error léxico durante la creación del objeto analizador sintáctico y, de ser así, desplaza la construcción de los analizadores al interior de la sentencia `try`.

7. Ampliación 1: Enteros largos

Con esta ampliación, se permite trabajar con enteros de longitud arbitraria. Bastará con modificar el analizador léxico.

Nivel léxico: Completa la especificación léxica de la sección 6.3 modificando la expresión asociada a la categoría **entero**. Ten en cuenta que, según la especificación del lenguaje, para los literales enteros se admite el carácter `0` o cualquier secuencia de dígitos que comience por uno distinto del `0`.

A partir de tu especificación, diseña e implementa la correspondiente MDD. Para ello, tendrás que modificar el método `siguiente` de la clase `Lexico`.

8. Ampliación 2: Paréntesis

Con esta ampliación, se pueden utilizar paréntesis en las expresiones. Bastará con modificar el analizador léxico para introducir las categorías **apar** (abre paréntesis) y **cpar** (cierra paréntesis) y el analizador sintáctico para permitir su utilización.

Nivel léxico: Completa la especificación léxica con las nuevas categorías y modifica la MDD.

Nivel sintáctico: La nueva gramática será la siguiente:

$$
\begin{aligned}
\langle \text{Línea} \rangle &\rightarrow \langle \text{Expresión} \rangle\ \textbf{nl} \\
\langle \text{Expresión} \rangle &\rightarrow \langle \text{Término} \rangle (\textbf{suma}\ \langle \text{Término} \rangle)^* \\
\langle \text{Término} \rangle &\rightarrow \langle \text{Factor} \rangle (\textbf{producto}\ \langle \text{Factor} \rangle)^* \\
\langle \text{Factor} \rangle &\rightarrow \textbf{entero} \mid \textbf{apar}\ \langle \text{Expresión} \rangle\ \textbf{cpar}
\end{aligned}
$$

Para actualizar la clase `Sintactico`, bastará con cambiar en `sintactico.py` el método `analizaFactor`. Lo que tendrá que hacer ahora el método será examinar `self.componente`. Si es de la categoría **entero**, deberá construir y devolver un objeto `NodoEntero`, como hasta ahora. Si se encuentra un componente de la categoría **apar**, tendrá que:

- avanzar el analizador léxico;
- llamar a `analizaExpresion`;
- comprobar que `self.componente` tiene un paréntesis cerrado, avanzando el léxico si es así o elevando una excepción en caso contrario;
- finalmente, debe devolver el árbol que haya recibido de `analizaExpresion`.

Si `self.componente` no está en ninguno de los casos anteriores, debe elevar una excepción. Observa que no necesitamos nuevos tipos de nodos en el AST.

9. Ampliación 3: Cadenas

Esta ampliación permite utilizar cadenas en las expresiones. Esto requerirá mayores cambios que las modificaciones anteriores. En el analizador léxico tendremos que añadir la nueva categoría **cadena**. Sintácticamente, tendremos que añadir otra posibilidad para ⟨Factor⟩. Además, definiremos nuevos nodos para los AST y tendremos que incluir comprobaciones de tipos.

Nivel léxico: Amplía la especificación léxica para incluir la clase **cadena** teniendo en cuenta lo que se dice en la sección 3.1. Piensa qué diferencias hay entre evitar la aparición de secuencias de escape incorrectas mediante la expresión regular o mediante la acción de cálculo del atributo correspondiente. En `componentes.py`, tendrás que añadir una nueva clase para esta categoría. En el constructor de esta categoría añade el código para eliminar las comillas que delimitan la cadena y sustituir las secuencias de escape por los caracteres que representan. Por ejemplo, el lexema `"El autor de \"El Quijote\" es Cervantes"` representa la cadena `El autor de "El Quijote" es Cervantes`. Modifica la MDD para reflejar la nueva especificación.

Nivel sintáctico: En la parte sintáctica, bastará con que la regla de ⟨Factor⟩ sea

$$\langle \text{Factor} \rangle \quad \rightarrow \quad \textbf{entero} \mid \textbf{cadena} \mid \textbf{apar}\ \langle \text{Expresión} \rangle\ \textbf{cpar}$$

Modifica `analizaFactor` para incorporar este cambio.

Nivel semántico: En `AST.py` habrá que añadir `NodoCadena` para representar los literales de cadena. Su estructura será similar a `NodoEntero`.

Hasta ahora, no hemos tenido problemas con los tipos de las expresiones. Sólo teníamos expresiones de un tipo, el entero. Con la introducción de las cadenas, la situación cambia. Por ejemplo, aunque la gramática lo permite, no es posible multiplicar dos expresiones si ambas corresponden a cadenas. Esto nos obliga a añadir un nivel adicional de comprobaciones. Añade a los nodos un nuevo método, `compsemanticas`, que no tendrá parámetros. Tras su ejecución, el nodo tendrá un atributo, `tipo`, con uno de los dos valores definidos en el módulo `tipos`. En `NodoEntero` y `NodoCadena`, `compsemanticas` se limita a asignar a `self.tipo` el valor correspondiente (`tipos.Entero` o `tipos.Cadena`). En `NodoSuma` y `NodoProducto`, `compsemanticas` comenzará llamando a los métodos `compsemanticas` de los hijos izquierdo y derecho del nodo. Para la suma, exigiremos que ambos hijos tengan el mismo tipo, que será el que se asigne a `self.tipo`. Si no tienen el mismo tipo, `compsemanticas` deberá elevar una excepción del tipo `ErrorSemantico`. En el caso del producto, el tipo del resultado será cadena si uno de los hijos es de tipo cadena y el otro entero. Si ambos hijos son de tipo entero, el producto también lo será. Finalmente, si ambos hijos son de tipo cadena, se elevará una excepción.

El programa principal deberá llamar a `compsemanticas` del nodo raíz del AST que ha encontrado antes de evaluarlo.

Generación de resultados: Dado que seguimos el convenio de Python, no hará falta modificar los métodos `evalua` de los nodos `NodoSuma` y `NodoProducto` para tener en cuenta las nuevas posibilidades.

10. Ampliación 4: Resto de operadores

Modifica tu intérprete para incluir todos los operadores de la sección 5. La gramática correspondiente sería:

$$\begin{aligned}
\langle \text{Línea} \rangle &\rightarrow \langle \text{Expresión} \rangle\ \textbf{nl} \\
\langle \text{Expresión} \rangle &\rightarrow \langle \text{Término} \rangle (\textbf{opad}\ \langle \text{Término} \rangle)^* \\
\langle \text{Término} \rangle &\rightarrow \langle \text{Factor} \rangle (\textbf{opmul}\ \langle \text{Factor} \rangle)^* \\
\langle \text{Factor} \rangle &\rightarrow \textbf{entero} \mid \textbf{cadena} \mid \textbf{apar}\ \langle \text{Expresión} \rangle\ \textbf{cpar} \mid \textbf{opad}\ \langle \text{Factor} \rangle \mid \textbf{barra}\ \langle \text{Expresión} \rangle\ \textbf{barra}
\end{aligned}$$

Observa que hemos agrupado los operadores aditivos (`+` y `-`) en la categoría **opad** y los operadores multiplicativos (`*` y `/`), en la categoría **opmul**.

Intenta decidir tú las modificaciones necesarias en los niveles léxico, sintáctico y semántico.