PROCESADORES DE LENGUAJES

# Práctica 3
# Minicalc, un intérprete para una calculadora sencilla.

Curso 2016/2017

# Índice

# 1. 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.

El resto del documento está organizado del siguiente modo. En la sección 2 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 sección 3 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. En la sección 4 se proponen las ampliaciones a realizar a la versión básica de la calculadora, y que ofrecen a la calculadora una mayor versatilidad de funcionamiento. Y finalmente, en la sección 5 se describen algunos detalles a tener en cuenta en la implementación, el formato de la memoria a presentar en esta práctica, y los detalles de evaluación de la misma.

Nota 1: Es recomendable asegurarse 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 2.1 para depurar errores.

Nota 2: Por otra parte, el desarrollo de la práctica es recomendable hacerlo en Linux, y obligatoriamente utilizando Python como lenguaje de programación. Por lo demás, puedes elegir libremente el editor o entorno de programación que más te guste.

# 2. Especificación

## 2.1. Invocación y comportamiento de la calculadora

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

`./minicalc [`*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 tildes) 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 deberán aparecer en caso de suceder (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.

## 2.2. Estructura de la entrada

La entrada de `minicalc` 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.

[1] Para poder ver los a árboles, puedes escribirlos en el formato de la herramienta `verArbol`, la cual la podéis descargar del campusvirtual de la asignatura, en la sección prácticas dentro del paquete `metacomp`, junto con un pequeño manual de funcionamiento.

Algunas expresiones en `minicalc` (más adelante se explica su significado) 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).

## 2.3. Algunos componentes léxicos

### 2.3.1. Literales

En `minicalc` 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.

### 2.3.2. Componentes léxicos que se omiten

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

## 2.4. Tipos de datos

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

## 2.5. Expresiones

En `minicalc` 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, y modificar la precedencia habitual de los operadores.

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.

# 3. Guía de desarrollo

## 3.1. 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.

### 3.1.1. Estructura de la calculadora

El fichero principal será `minicalc` 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. Se proporciona una versión inicial de este fichero.

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`, el cual también se proporciona.

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 `arboles.py` (también proporcionado).

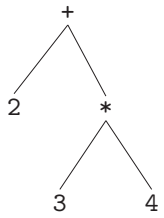
Finalmente, tendremos dos ficheros auxiliares: `errores.py` y `tipos.py`. El fichero `errores.py` simplemente contendrá cuatro clases derivadas de `Exception`: `ErrorLexico`, `ErrorSintactico`, `ErrorSemantico` y `ErrorEjecucion`. Ambos archivos también se proporcionan.

El módulo `tipos.py` define dos variables: `Entero` y `Cadena`. Para `minicalc`, 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 ésta, el semántico tiene un papel muy reducido.

### 3.1.2. 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:




El módulo `arboles.py` también se proporciona. En él, podemos ver que 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, cada nodo define 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, primero se evalúa el hijo izquierdo, después el derecho y, finalmente, se suman o multiplican los valores para devolver el resultado.

También se proporciona el método `__str__` en todos los nodos, el cual devuelve una representación "agradable" del objeto.

Para implementar la opción `-s`, deberás añadir un método llamado `arbol`. Este método no tiene parámetros y devuelve una representación en el formato verArbol. Por ejemplo, para las sumas y productos devolverá una cadena formada por: un paréntesis abierto, el operador entre comillas, el resultado de llamar al método `arbol` del hijo izquierdo, el resultado de llamar al método `arbol` del hijo derecho y un paréntesis cerrado (ver el manual de verArbol para más información).

### 3.1.3. 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`<br>emitir |
| **cadena** | `"[^"\n]"` | `valor` | calcular `valor`<br>emitir |
| **abre** | `\(` | `__` | emitir |
| **cierra** | `\)` | `__` | emitir |
| **blanco** | `[␣\t]` | `__` | omitir |
| **nl** | `\n` | `__` | emitir |
| **eof** | `eof` | `__` | emitir |

Por un lado, están los operadores y los paréntesis; por otro, los literales enteros y de cadena. 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**.

### 3.1.4. Implementación del analizador léxico

Vamos a descomponer el análizador léxico en dos partes: la definición de los componentes léxicos y el analizador en sí. El módulo `componentes.py` se proporciona, y en él se define una clase por cada

categoría de las que no se omiten. Cada clase Python tiene un atributo `cat`, que nos indica a qué categoría pertenece el componente léxico[2] . Además, las clases **entero** y **cadena** tendrán el atributo `valor` que se calcula en el constructor a partir del lexema.

Para poder comprobar el funcionamiento del analizador, es de gran ayuda poder escribir cómodamente los componentes léxicos. Para ello, se ha añadido a cada objeto el método `__str__`, en el que hemos tomado la convención de escribir un componente mediante el nombre de su categoría y, en el caso de los enteros y las cadenas, mediante el nombre seguido de la cadena "`valor:`" y el valor entre paréntesis. De este modo, con

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

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

Ahora debemos definir una clase para representar el analizador léxico. Para ello, creamos en el fichero `lexico.py` la clase `Lexico`. El comportamiento de los objetos de esta clase es 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 (a modo de programa de prueba para el analizador léxico):

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

deberá escribir por pantalla:

```
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.

El fichero `lexico.py` se proporciona. En el podemos ver el constructor de la clase `Lexico`. Como parámetro, recibe una cadena que representa la línea que contiene la expresión (en este caso implementado en la clase `flujo`), y un atributo `poserror` para guardar la posición de la cadena en la que se pueda producir un eventual error léxico. El constructor de la clase `flujo`, además de copiar la cadena, inicializa a -1 un atributo (`pos`) que indica a partir de qué posición de la cadena esperamos encontrar el siguiente componente. El método, `siguiente`, implementa una MDD para la especificación léxica dada. Cada vez que se llama a este método sin parámetros y que al ser llamado devuelve el siguiente componente léxico que se encuentre en la cadena y que no haya que omitir. Si se encuentra un error (un carácter que no esté en alguna de las categorías léxicas definidas), eleva una excepción de tipo `ErrorLexico`.

La implementación del analizador léxico se ha hecho sin utilizar la librería `re` de Python. Piensa en cómo realizarías esta implementación utilizando esa librería, pues en la versión final tendrás que aportar esta solución.

### 3.1.5. 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 GPDR:

[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.

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

¿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 o cadena, o una expresión entre paréntesis.

En el fichero `sintactico.py` se proporciona una implementación de un analizador sintáctico para esta gramática, que construye el correspondiente AST para cada línea/expresión de entrada. Para ello, se ha creado 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`, para tener ya disponible el primer componete léxico de la entrada.

La parte principal de `Sintactico` es un conjunto de métodos que se corresponderán con los no terminales de la gramática. Estos métodos son `analizaLinea`, `analizaExpresion`, `analizaTermino` y `analizaFactor`. 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 paréntesis abierto, en cuyo caso avanza en el léxico, analiza la expresión mediante `analizaExpresion` y después comprueba que la categoría de componente es igual a `cierra`. Si todo ha ido bien devuelve el árbol correspondiente. En caso contrario eleva una excepcion de tipo `ErrorSintactico`.
- un entero o una cadena. Si no, eleva una excepción de tipo `ErrorSintactico`. Si efectivamente hay un entero/cadena, crea un árbol con `NodoEntero`/`NodoCadena` utilizando el valor que tiene el componente y, tras avanzar en el léxico, devuelve ese árbol.

### 3.1.6. Comprobaciones semánticas

Hasta ahora, no hemos comentado nada con respecto a los tipos de las expresiones. 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. Como puedes observar los nodos contienen

un método llamado `compsemanticas`, que no tiene parámetros. Tras su ejecución, el nodo tendrá un nuevo atributo, `tipo`, con uno de los dos valores definidos en el módulo `tipos.py`. 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.

La realización de las comprobaciones semánticas de cada expresión se realizar en el programa principal (explicado a continuación), donde llama a `compsemanticas` del nodo raíz del AST que ha encontrado antes de evaluarlo, siempre y cuando no haya habido errores léxicos y/o sintácticos.

#### 3.1.7. Programa principal

Finalmente, el programa principal, que llamamos `minicalc.py` importa los módulos necesarios, e implementa un bucle que hace 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 correspondiente mensaje de error según el formato especificado en la página 1.
- Si no ha habido errores, escribe, por `stdout`, el resultado de evaluar la expresión.

Nota 3: 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`.

Nota 4: El tratamiento de las opciones del programa no se proporciona. Esto es lo único que deberéis incluir en este programa.

# 4. Ampliaciones a desarrollar

A continuación se proponen una serie de ampliaciones que ofrecen mayor versatilidad al funcionamiento de la calculadora.

## 4.1. Ampliación 1: Cadenas

La especificación actual de la categoría léxica `cadena` no permite incluir sencuencias de escape dentro de la propia cadena. La idea de esta ampliación es modificar la especificación léxica de la clase `cadena` teniendo en cuenta lo que se dice en la sección 2.3.1. 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 el constructor de esta categoría habrá que añadir 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. También habrá que hacer lo propio en la implementación mediante expresiones regulares.

## 4.2. Ampliación 2: Resto de operadores

Modifica tu intérprete para incluir todos los operadores de la sección 2.5 . La gramática correspondiente sería:

⟨Línea⟩ → ⟨Expresión⟩ **nl**
⟨Expresión⟩ → ⟨Término⟩ (**opad**⟨Termino⟩)*
⟨Término⟩ → ⟨Factor⟩ (**opmul**⟨Factor⟩)*
⟨Factor⟩ → **entero** | **cadena** | **apar** ⟨Expresión⟩ **cpar** | **opad** ⟨Factor⟩ | **barra** ⟨Expresión⟩ **barra**

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.

### 4.3. Ampliación 3: Literales de tipo real

Con esta ampliación se pretende permitir a la calculadora trabajar con literales reales, lo cual requerirá realizar cambios a todos los niveles: léxico, sintáctico y semántico.

La calculadora deberá permitir realizar operaciones entre operandos de distinto tipo (entero y real), promocionando el resultado siempre al tipo más general, en este caso el real. El comportamiento del operador **barra** con los reales tendrá el mismo signicado que para los enteros. La multiplicación de un real por una cadena se deja abierta al alumno, pero deberá tenerse en cuenta y por tanto que esa operación tenga un significado claro y preciso, que habrá que tener en cuenta en la implementación.

## 5. Aclaraciones para la realización y evaluación de la práctica

### 5.1. Algunos detalles sobre la especificación e implementación

Deberán implementarse dos versiones del analizador léxico: una utilizando la biblioteca `re` de Python y otra sin utilizarla.

Al implementar las opciones `-s` y `-a` se debe conseguir que a partir de siguiente comando (en Linux, claro está):

```
$ cat prueba | ./minicalc -s | ./verArbol
```

se visualice el AST de cada una de las líneas contenidas en el fichero prueba. Y cambiando `-s` por `-a` se debe visualizar el árbol de derivación en vez del AST.

La implementación de la opción `-a` no se explica detalladamente en la guía de desarrollo. Eso es intencionado. Pensar cómo hacerlo es un ejercicio aconsejable para entender mejor el análisis sintáctico. Es recomendable no preguntar cómo se hace antes de intentar resolverlo sin ayuda (pero pregunta después, si hace falta).

En esta asignatura debemos distinguir entre "letra", con lo que nos referimos a cualquiera de las 26 letras del alfabeto latino, y "carácter", con lo que nos referimos a cualquiera de los caracteres ASCII imprimibles junto con el espacio, el tabulador y el fin de línea. Basta con que `minicalc` acepte entradas escritas utilizando esos caracteres, no es necesario que acepte caracteres representables con otras normas de codificación. En los ficheros de prueba nos limitaremos a utilizar esos caracteres.

El enunciado de la práctica tiene dos secciones principales: especificación y guía de desarrollo. La especificación describe lo que debes implementar respetándola al pie de la letra. Puedes verla como la descripción de los requisitos de un cliente para el que estás desarrollando la calculadora. La guía de desarrollo contiene sugerencias sobre cómo implementar la calculadora, pero no tienes por qué seguirla al pie de la letra, puedes adoptar tus propios criterios en todo aquello que consideres conveniente (nombres de variables, división del código en clases y ficheros, algoritmos empleados, etc.) teniendo siempre presente que tu implementación debe ser adaptable para incorporar nuevas ampliaciones.

Uno de los requisitos de la especificación es que la calculadora se pueda ejecutar escribiendo en un terminal la orden `./minicalc [`*`opción`*`]`, no `python minicalc.py [`*`opción`*`]` ni `./minicalc.py [`*`opción`*`]`. Si no te funciona, quizás te falte hacer algo de esto:

- `$ mv minicalc.py minicalc`
- `$ chmod +x minicalc`
- Añadir `#!/usr/bin/env python` al principio del fichero `minicalc`.
- `dos2unix minicalc`, si se ha escrito en un editor para Windows.

Es importante comentar que la evaluación final (y automática) de la prácticas se hará en un entorno Linux, concretamente en una distribución Ubuntu 12.04 y con la versión 2.7 de Python.

## 5.2. Memoria a presentar

Todo el trabajo desarrollado en esta prácticas deberá documentarse adecuadamente en una memoria final que deberá estar bien estructura y redactada.

A modo de ejemplo un posible índice de la memoría podría ser:

1. **Introducción.**
2. **Análisis léxico:**
   a) Diagrama de la correspondiente máquina discriminadora determinista. Acciones asociadas a los estados de la misma.
   b) Tratamiento de los errores léxicos. Errores detectados en este nivel y técnicas de recuperación utilizadas.
   c) El analizador léxico. Cuestiones de implementación: gestión de la lectura y devolución de caracteres; componentes léxicos; tratamiento de los blancos, etc. Organización del código correspondiente al analizador léxico. Descripción de sus principales elementos (clases, objetos, variables...).
3. **Análisis sintáctico:**
   a) Tabla de análisis RLL(1). Proceso de construcción de la tabla de análisis: primeros, siguientes, etc.
   b) Tratamiento de los errores sintácticos. Errores detectados en este nivel y técnicas de recuperación utilizadas.
   c) El analizador sintáctico. Cuestiones de implementación: comunicación con el analizador léxico; tratamiento de errores, etc. Organización del código correspondiente al analizador sintáctico. Descripción de sus principales elementos (clases, objetos, variables...).
4. **Análisis semántico:**
   a) Relación de comprobaciones semánticas. Errores detectados en este nivel y su tratamiento.
   b) El analizador semántico. Descripción de la clase AST. Cuestiones de implementación, organización del código correspondiente y descripción de sus elementos principales.
5. **Intérprete**: descripción del intérprete, implementación, etc.
6. **Descripción de casos de prueba**: Descripción de los casos de prueba usados (y propuestos) para comprobar el funcionamiento de la calculadora.
7. **Ampliaciones propuestas**: En este apartado para cada ampliación propuesta se deberá proporcionar una sección que deberá constar de las siguientes partes:
   a) Descripción esquemática de los pasos a llevar a cabo con dicha modificación. Se deberá hacer a todos los niveles: léxico, sintáctico, semántico, y de generación de resultados, si los hubiere.
   b) Partes del código fuente modificado de la implementación base. No listados de código, simplemente las partes modificadas, con explicación de cada una de ellas.
   c) Casos de prueba relacionados con la modificación.
8. **Valoración personal**: En este apartado, opcional, se intentará hacer una valoración de los aspectos que se consideren oportunos: idoneidad con respecto a los contenidos teóricos, nivel de dificultad, tiempo dedicado, etc.

Dada la experiencia de años anteriores, es importante saber que no se trata de volver a copiar el contenido de este documento en ésta memoria de prácticas, sino precisamente todo lo contrario, en ella deberá aparecer todo lo que no se muestra en este documento. Esto redundará en un ahorro sustancial de tiempo y espacio, así como en la presentación de una memoria clara y centrada en los aspectos relevantes del desarrollo de la práctica.

## 5.3. Evaluación

La entrega de esta práctica es obligatoria, tanto para los alumnos que opten por la evaluación continua como los que no. Con ello aseguraremos que el alumno ha conseguido los conocimientos mínimos para poder llevar a cabo con éxito el estudio y la implementación de un intérprete sencillo, pero que cubre con todos los aspectos teórico-prácticos expuestos hasta la fecha.

La fecha de entrega de cada parte se anunciarán en la plataforma virtual con la suficiente antelación, pero será durante la semana 10 del curso (actualmente estamos en la semana 7).

En la evaluación de la práctica se tendrá en cuenta:

- La calidad de la memoria presentada.
- El funcionamiento del intérprete: para ello se utilizarán muchos casos de prueba (muy sencillos pero que contemplen todas las características del lenguaje de la calculadora), y se realizará de formas automática mediante un oráculo. Por este motivo es IMPORTANTÍSIMO seguir al pie de la letra las especificaciones de funcionamiento y los detalles comentados en la sección 5.1.
- La nota obtenida en esta prácticas corresponderá a un 20 % de la nota final de la asignatura.