Representación de Código

Para poder trabajar con código Lox de una manera eficiente, es necesario poder tener una representación de este código dentro del lenguaje donde escribamos un compilador.

El código del lenguaje Lox tiene una estructura, que está definida por una gramática libre de contexto. Nuestro objetivo en está práctica es preparar una representación del código de Lox dentro del fichero *Representacion.py*.

Como Python permite la programación orientada a objetos, vamos a utilizar clases para representar código Lox. Demos un extracto del código que debemos generar:

```
from dataclasses import dataclass
from Lexer import Token
from typing import List, Optional

@dataclass
class Declaration:
    pass

@dataclass
class Primary:
    pass



@dataclass
class Unary:
    op: str
    atr: Optional["Unary"]=None  # Esto es para representar call o otro Unary

@dataclass
class Call:
    base: Primary

@dataclass
class Number(Primary):
    tok: Token

@dataclass
class Factor:
    op: str
    first_un: Unary
    second_un: Unary



@dataclass
class Function:
    name: str
    params: List['Parameter']
    body: 'Block'

@dataclass
class ClassDeclaration(Declaration):
    name: str
    father: str
    methods: List[Function]

@dataclass
class FunctionDeclaration(Declaration):
    fun: Function

@dataclass
class VarDeclaration(Declaration):
    name: str
    expr: 'Expression'

@dataclass
class Statement(Declaration):
    pass

@dataclass
class Program:
    declarations: List[Declaration]
```

Notesé que las clases coinciden con el nombre de las distintas variables y, por convención, el nombre

de las clases empieza siempre por mayúscula. 1) *Se pide completar las clases, según la gramática de Lox.*

Es conveniente añadir un método a las clases, de forma que se imprima la representación del objeto. Por ser esta representación un árbol, es conveniente que la salida este correctamente indentada y por ello el método tiene que tomar un argumento con el número de espacios de indentación.

```
@dataclass
class Number(Primary):
    tok: Token
    def tostring(self, n):
        output = " " * n + self.tok.tipo + "\n"
        output += " " * (n + 2) + self.tok.value  # Aquí ponemos el valor un poco más indentado
        return output

@dataclass
class Factor:
    op: str
    first_un: Unary
    second_un: Unary

    def tostring(self, n):
        output =""
        output += " "*n + self.op + "\n"
        output += self.first_un.tostring(n+2) + "\n"
        output += self.second_un.tostring(n+2) + "\n"
        return output
```

Con este código se puede realizar la siguiente llamada:

```
Factor(op="/", first_un=Number(tok=Token(lineno=0, value="10", tipo="Int"))
       ,
       second_un=Number(tok=Token(lineno=0, value="10",tipo="Int")))
.tostring(2)
```

y obtener la siguiente salida

```
/
  Int
    10
  Int
    10
```

2) *Corríjase la clase Factor, de forma que siga la gramática y añádase el método tostring(self, n) a todas las clases realizadas en el apartado anterior.*