Mini Proyecto ED 1 v1 – El parser

¿Os habéis preguntado alguna vez cómo se hacen los IDEs (*Integrated Development Environment*), como el que se utilizará en el curso, para identificar si hace falta o sobra un paréntesis o llave, o si bien nos equivocamos (por ejemplo, al escribir un paréntesis cuando tocaba una llave)? Este problema es un clásico en Estructuras de Datos. Podemos resolverlo con una solución de orden lineal O(N) utilizando un tipo de Estructura de Datos concreta: una **pila**. En este mini proyecto, nos centraremos justamente en esto: diseño e implementación de un **parser**.

Ejercicio: diseñar e implementar un parser

El problema que resuelven los IDEs es algo más complejo que lo que os proponemos, ya que ellos parten de caracteres de un texto y lo dividen en unidades más pequeñas denominadas **Tokens** (por ejemplo, un número, un nombre, un paréntesis abierto, un punto y coma, etc.). Nosotros partiremos del trabajo hecho por el Tokenizer y supondremos que éste nos ha dejado su resultado en una lista de Tokens, que sólo contiene la información que nos interesa: paréntesis y llaves, tanto abiertos como cerrados.

El objetivo principal de este mini proyecto de Estructura de Datos es diseñar e implementar un método tal que, dada una lista de elementos de tipo Token, nos permita identificar si la expresión formada por los Token es correcta en un tiempo de ejecución de **orden lineal**. Ejemplos de expresiones correctas e incorrectas son:

1) ( ) Correcta
2) ((Incorrecta
3) ({ }) Correcta
4) ({ (} Incorrecta

¿Podéis pensar por un momento por qué las expresiones (1) y (3) son correctas? Fijaos en las parejas de tokens (abierto y cerrado), y en el orden de apertura y de cierre.

Como en la asignatura seguimos el paradigma de POO, nos gustaría tratar cada token como si fuera un objeto. Concretamente, y para simplificar, tendremos los siguientes tokens:

- `ClosingCurly -> }`
- `ClosingParenthesis –> )`
- `OpenCurly –> {`
- `OpenParenthesis –> (`

Para que el Parser pueda trabajar sobre los Tokens, definiremos una interfaz que cada uno de estos debe implementar. De esta manera, nuestro Parser será independiente de las clases concretas que implementen la interfaz Token.

¿Qué se ha de hacer?

Básicamente se han de realizar dos tareas. La primera es la implementación de:

- `Token` (interfaz). Esta interfaz debe contener los métodos que necesitamos para implementar Parser. Por ejemplo: debemos saber si un Token es de apertura, cierre y/o si

forma pareja con otro. Fijaos en que las operaciones que hemos de incluir en la interfaz han de ser la que necesite el algoritmo de comprobración.

- `ClosingCurly, ClosingParenthesis, OpenCurly, OpenParenthesis:` serán clases que implementarán la interfaz `Token`.

La segunda tarea consiste en implementar el método **parse** dentro de una clase llamada Parser. Este método recibe como parámetro un `List<Token>` y devuelve *cierto*, si la expresión es correcta y *false*, en caso contrario.

```
public static boolean parse(List<Token> tokens)
```

Fijaos que los elementos de la lista son de tipo `Token`. Por lo tanto, el parámetro puede ser una combinación de `Closing/OpenCurly, Closing/OpenParenthesis`, ya que **todos** son Tokens.

Como hemos comentado antes, este problema se puede resolver utilizando una pila y, concretamente, usaremos la clase Stack <T> de JCF.

Estrategia para resolver el problema: considerar que hay tokens (**abiertos/cerrados**) y (**paréntesis/llaves**). La idea es recorrer la lista de tokens de izquierda a derecha. Si nos encontramos con un token de apertura, lo hemos de guardar, para poder comprobar posteriormente que hemos encontrado el de cierre. ¿Y si encontramos uno de cierre? Para que todo sea correcto, este debería cerrar el último token de apertura encontrado. Si llegamos al final de la lista sin encontrar paréntesis o llaves sin pareja, podemos decir que la expresión es correcta.

Entrega

Un proyecto Netbeans (comprimido, con el nombre de los integrantes del grupo) con:

- Clases/Interfaces: Token, ClosingCurly, ClosingParenthesis, OpenParenthesis, OpenCurly.
- El método parse definido en el fichero Parser.java
- Definir varias funciones que comprueben el buen funcionamiento del método parse en varios casos. Pensad en diferentes casos positivos / negativos. Recordad, cuantas más cosas hayáis probado que funcionan correctamente, más difícil será que vuestra solución contenga errores.
- Documento de texto con la estrategia de la solución implementada en parse (es decir, pseudocódigo y utilización de la Pila).

Recordatorio: una entrega por grupo.

Criterios de evaluación y calificaciones

- No entrega / Entrega fuera de la fecha establecida **= 0**
- Entrega parcial:
    - Entrega, pero con errores de compilación y/o ejecución **<=4**
    - Entrega, pero falta alguna de las partes **<= 2**

- Entrega completa: funcional y sin errores **5-10**

El profesorado se reserva el derecho de aplicar criterios de corrección, como a la hora de tener en cuenta la calidad y elegancia del código (por ejemplo, nombre de las variables, variables innecesarias o que no se utilicen, etc.), aplicación de diseño descendente (es decir, dividir problemas grandes en pequeños, y delegar estos en funciones), diseño orientado a objetos, comentarios en el código que clarifiquen, pruebas adicionales, entre otros, con el fin de asignar la nota del laboratorio.

Posibles ampliaciones

- Ampliación de vuestra solución con otro tipo de parejas: [i]. ¿Qué es necesario cambiar?
- Implementad una función que tome un texto del usuario (String) y compruebe que representa una expresión correcta entre paréntesis (los caracteres que no son paréntesis ni llaves se pueden ignorar).
- Haced lo mismo, pero leyendo los caracteres (uno a uno) de un archivo de texto.