Programación semiautomática de un analizador léxico

Objetivo: construir analizadores léxicos generales, a falta de los detalles propios de cada lenguaje (los detalles se fijarán en la función main).

Esos “detalles” incluyen, como mínimo:

- Un autómata para reconocer el lenguaje
- El alfabeto del lenguaje (a bajo nivel las letras son números)
- Funciones para traducir de cadenas del alfabeto a vectores de números enteros y viceversa
- Tabla que establezca la relación entre estados finales del autómata y los identificadores de tokens.

Programación semiautomática de un analizador léxico

//Clase autómata finito

/*

Sus objetos son autómatas finitos con un número "numEstados" de estados

indexados desde 0 hasta numEstados-1. Fijamos que el estado 0 sea siempre el inicial.

Utilizamos un entero "tamAlfabeto" para representar el número de caracteres del alfabeto. Los caracteres serán también enteros indexados desde 0 en adelante.

Podemos representar los estados finales con un vector de booleanos o un *Set de enteros*. Se puede utilizar otro atributo para indicar el estado actual

*/

```
public abstract class AutomataFinito {

        private int numEstados;
        private boolean[] finales;
        private int tamAlfabeto;
        // private int estActual;
```

Programación semiautomática de un analizador léxico

//Clase autómata finito

```
//Constructores: sin finales (luego se pueden marcar) y con finales
        public AutomataFinito (int num, int alfabeto)
        public AutomataFinito (int num, int alfabeto, boolean [] finales)
//Otros métodos
                marcarFinal(int estado)
                setFinales(boolean[] estadosFinales)
                getNumEstados()
                getFinales()
        //Método observador para determinar si un estado es final
                esEstadoFinal (int estado)

/*Método de transición. Se puede pensar como método que informa del estado que se alcanza
partiendo de un estado y recibiendo una entrada*/
        public abstract int transicion (int estado, int letra)

/*Otra versión, pensando que una transición produce una modificación del estado actual del
autómata. Resulta mejor como orientación a objetos, pero complica un poco nuestro uso. */
        public abstract void transicion (int letra)

//La extensión de la transición a cadenas. Posible versión void si transición modifica el estado
                ??cierreTransicion (int estado, int cadena [])

//Método que informa si una cadena pertenece al lenguaje definido por el autómata
                ???  perteneceLenguaje (???)
```

Programación semiautomática de un analizador léxico

// Implementación de autómata finito usando una matriz de transiciones

public class AutomataFinitoMatriz extends AutomataFinito {

/* La transición se representa por una matriz. Las filas representan estados, las columnas letras. La entada (i, j) de la matriz será el estado alcanzado desde el estado ”i” si se recibe la letra “j”.
Podemos poner un -1 para indicar error, esto es, no existe esa transición
*/

private int [] [] matriz;

//Constructores

// Implementación de la función de transición. Devuelve la correspondiente entrada de la matriz. Habrá que ser consecuente si se usa la “versión void”. */

public int transicion (int estado, int letra)

}

Programación semiautomática de un analizador léxico

```
//Clase Token.
//Una clase cuyos objetos son parejas (token, lexema)

public class Token {
        //En realidad l estamos llamando token a la pareja (token, lexema)
        private String idToken;
        //Para el lexema se puede usar un vector de enteros, List de entero,….
        private --------lexema;

        public Token(String id, -----  lexema) {}
        public String getId() {}
        public  -----   getLexema() {}
        }
```

Programación semiautomática de un analizador léxico

// La clase analizador léxico contiene los métodos para hacer el análisis léxico de //una cadena

//Clase AnalizadorLéxico.

// La clase analizador léxico contiene los métodos para hacer el análisis léxico de una cadena

/*

El estado de un analizador léxico lo determina:

- la cadena analizar (y un índice que indique hasta donde ya ha sido analizada)
- el autómata encargado de hacer el análisis
- una estructura que relacione estados finales del autómata con identificadores de tokens. Un Map puede ser adecuado
- un histórico de los tokens ya obtenidos en el trozo de cadena ya analizada

*/

Programación semiautomática de un analizador léxico

```
public class AnalizadorLexico {
        private int[] cadena;
        private AutomataFinito A;
        private int posActual;
        private Map<Integer, String> tokens;
        private List<Token> historico;
```

/*El constructor dará valor a los atributos mediante los parámetros e inicializará el análisis léxico de la cadena, es decir, no hay nada en el histórico y .posActual = 0 */

/*Método esencial para hacer avanzar el análisis; su función es generar el siguiente token. No hay que olvidar que debe ser el más largo posible.

Hay que ser muy cuidadoso para evitar errores por final de cadena, transición a error, etc.*/

```
        public Token nextToken() {}
```

/*

Programación semiautomática de un analizador léxico

```
public class AnalizadorLexico {
        private AutomataFinito automata;
        private int[] cadena;
        private int posActual;
        private Map<Integer, String> tokens;
        private List<Token> historico;
```

//Otros métodos necesarios:

- Método que informa de si quedan tokens por extraer (hasMoreTokens)
- Método que proporciona el histórico de tokens
- Método que reinicia el análisis (reset)
- Método que cambia la cadena a analizar, reiniciando el análisis actual (nuevaCadena)
- Método que proporciona todos los tokens, en vez de ir uno a uno (finalizarAnalisis)

Programación semiautomática de un analizador léxico

Clase AnalizadorLexico

```
private AutomataFinito automata;
private int[] cadena;
private int posActual;
private Map<Integer, String> tokens;
private List<Token> historico;
```

```
public Token nextToken() {
/*utilizar variables “locales” para:
• Anotar la posición en que estamos en cada instante en la cadena, sin modificar posActual
• Anotar el último estado final visitado
• Anotar en qué posición de la cadena se estaba en el último final visitado
*/
/*Con esos datos, se puede calcular el identificador del token y su lexema.
No se os olvide actualizar posActual al generar el token
*/
```

Programación semiautomática de un analizador léxico

Clase Main

//Es un ejemplo que no corresponde a nuestro caso

```
int matriz[][] = {{1, 2,4}, …,{-1, 4,7}};
int tamAlfabeto = 3;
int NumEstados=9;
boolean finales[] =    ;

String palabra=“acccbbaaba”

Map<Integer, String> equivTokens = new HashMap<>();
equivTokens.put(1, "cero");
Automata A=new…....
AnalizadorLexico escaner = new……;
```

//Métodos para traducción directa e inversa de cadenas sobre {a,b,c} a vectores de int

//Usar los métodos de AnalizadorLexico