CUADERNO PRÁCTICAS TEORÍA DE LA COMPUTACIÓN

ÁLVARO FRAIDIAS MONTEAGUDO

Índice:

Como lanzar las prácticas desde una clase java externa:

1º Compilamos el archivo.flex con la herramienta proporcionada por el profesor. Esto generará un archivo.java

2º Compilamos el archivo.cup. Para ello descargamos la herramienta CUP proporcionada por el profesor y la guardamos en la misma carpeta que el archivo.cup.

Una vez hecho esto ejecutamos la terminal y nos metemos en la ruta donde están estos archivos y ejecutamos la siguiente instrucción: java -jar java-cup-11b.jar <archivo.cup>

Si todo va bien nos generará dos archivos: El parser.java y el sym.java

3º Creamos un proyecto java y en la carpeta src pegamos los 3 archivos generados en los dos pasos previos. Una vez hecho esto, en el main ponemos:

```
Analizador analizador = new Analizador (new
FileReader (args[0]));
analizador.yylex();
```

Saldrán más de una bombilla con un círculo rojo. En ese caso vamos siguiendo los consejos que nos da NetBeans o Eclipse.

4º Añadir las librerías. Para hacer esto nos dirigimos a la parte de la izquierda de la pantalla, donde nos aparecen los proyectos, le damos clic derecho encima de las librerías y nos saldrá la opción “Add library ” y ahí añadimos los .jar del JFLEX y CUP.

5º Por último, añadir el texto.txt. Para ello nos vamos a la parte de arriba de NetBeans o Eclipse al apartado “RUN” y dentro de este apartado vamos a “Set Project Configuration” y allí añadimos el texto.txt .

Práctica 2

Esta práctica consiste en leer de un fichero todos los símbolos y categorizarlos dependiendo de los tokens definidos en otro archivo con la extensión .flex.

Las expresiones regulares que definen los tokens, que en mi caso he declarado 10, son los siguientes:

1º Un token que he declarado con el nombre "nombre" que engloba todos aquellos símbolos que cumplan la expresión regular:
`[A-ZÁÉÍÓÚ][A-Za-zÁÉÍÓÚáéíóú]*`

2º Un token con el nombre " número de teléfono" que engloba todos aquellos símbolos que empiecen con un "+" seguido de cualquier combinación de números (clausura de todos los números comprendidos entre 0 y 9). Su expresión regular es:
`[+34][6]?[7]?[0-9]*{8}`

3º Un token con el nombre "cifra" que a diferencia de " número de teléfono" este engloba los símbolos que sean una clausura desde el 1 al 9.
Su expresión regular es: **`[0-9]*`**

4º Un token que he declarado con el nombre "correo" que abarca todos aquellos símbolos que cumplan con la expresión regular:
`[_a-z0-9-]+(.[_a-z0-9-]+)*@[a-z0-9-]+(.[a-z0-9-]+)*(.[a-z]{2,4})`

5º Un token con el nombre "contraseña" que engloba todas las palabras alfanuméricas que están dentro del símbolo "&" incluido este símbolo.
Su expresión regular es: **`[&(a-zA-Z0-9)&]*`**

6º Un token **`"["`** que se utiliza para saber cuándo empieza una parte del texto que puede ser Cargos, Departamentos o Trabajadores.

7º Un token **`"]"`** que se usa para saber cuándo finaliza una parte del texto.

8º Un token **`";"`** que sirve para separar distintos tokens dentro del fichero.

9º Un token **`","`** que también sirve para separar distintos tokens dentro del fichero.

10º Los tokens **`"\n" "\t" "\s" "\r"`** que se usan para realizar saltos de línea dentro del fichero.

A parte de estos 10 tokens he declarado 3 palabras reservadas que son:
"Cargos", "Departamentos" y "Trabajadores".

Práctica 4

En esta práctica se utilizará también la herramienta cup para hacer un análisis sintáctico a parte del análisis léxico.

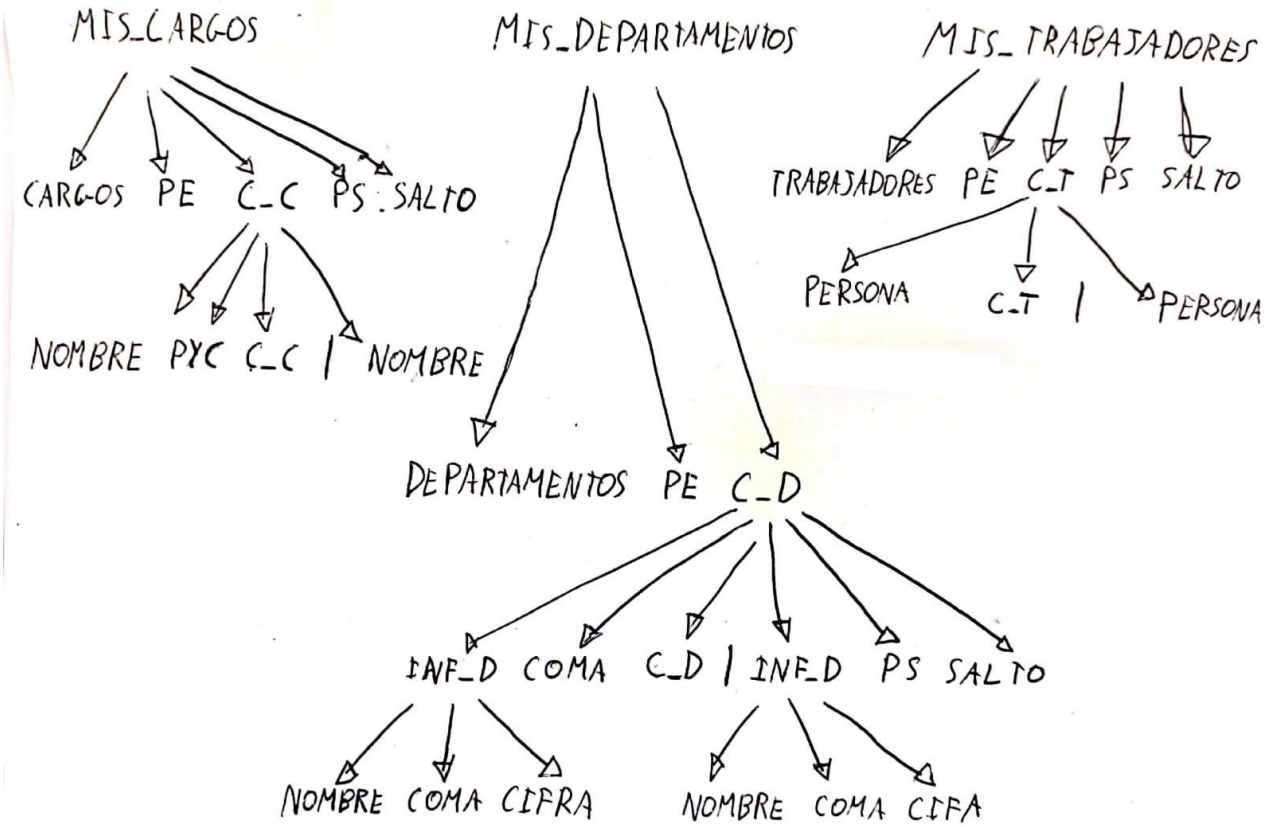
Primero debemos encontrar todos los tokens con el analizador léxico Jflex y utilizarlos en el analizador sintáctico cup como terminales para crear las reglas sintácticas.

La herramienta cup se encarga de recibir los tokens proporcionados por Jflex y crear una gramática que defina un lenguaje. Para ello hay que declarar la gramática en el archivo .cup y para eso usaremos un metalenguaje empleado para definir lenguajes libres de contexto, es decir, una manera formal de describir lenguajes formales. Este metalenguaje es EBNF.

Una vez definido esta gramática y recogidos los tokens mediante Jflex, lanzaremos el programa mediante una clase externa.

Los tokens que nos proporciona Jflex serán los símbolos terminales y los que creemos nosotros como pasos intermedios en la definición de la gramática serán los símbolos no terminales.

La definición de la gramática es la siguiente:



Donde:

*PE = PAR_ENT.

*PS = PAR_SAL.

*C_C = CONJUNTO_CARGOS.

*C_D = CONJUNTO_DEPAR.

*C_T = CONJUNTO_TRABAJ.

Práctica 5

En esta práctica aparte de hacer el análisis sintáctico y léxico también añadiremos 4 funcionalidades.

*La primera es darle al programa cierta libertad a la hora de afrontar un error sintáctico ya que antes si encontraba uno, finalizaba el programa. Ahora nos avisará sobre él y para ello he usado la palabra reservada "error" para que vaya saltando tokens hasta encontrar el token especificado a su derecha en la declaración de la gramática. Ejemplo:

```
CONJUNTO_TRABAJ ::= PERSONA CONJUNTO_TRABAJ |PERSONA|
error SALTO | PAR_SAL;
```

Si en CONJUNTO_TRABAJ se declara un error de sintaxis, irá saltando tokens hasta encontrar un salto de línea.

También lo he implementado en la definición de la gramática de cargos y departamentos.

*La segunda es comprobar que los cargos y departamentos no están repetidos y para ello he creado dos arrays, uno para los cargos y otro para los departamentos, donde iremos añadiendo cada vez que leamos un cargo o departamento. Antes de añadir uno u otro tendremos que comprobar que efectivamente no está repetido y para ello ponemos una función dentro de "parser code" para implementar esta funcionalidad.

La función es la siguiente:

```
private void nuevoCargo(String cargo){
     if (listaCargos.contains(cargo)){
          System.out.println("Aviso: cargo " + cargo + "
                              repetido.");
          }
          else{
               listaCargos.add(cargo.trim());
          }
     }
```

En caso de estar el cargo ya en el array, nos informará por pantalla que ya está repetido. (En el caso de departamento, la función es igual cambiando cargo por departamento)

Esta función por sí sola no haría nada si no la llamamos desde la gramática cada vez que leemos un cargo o un departamento. La gramática con la implementación de esta función es la siguiente:

```
CONJUNTO_CARGOS ::= NOMBRE:cargo {: nuevoCargo(cargo); :}
PYC CONJUNTO_CARGOS |NOMBRE:cargo {: nuevoCargo(cargo); :};
```

Cuando aparezca un cargo, llama a esta función y añade o no el cargo.

En el caso de departamento sería así:

```
INFORMACION_DEPAR::=
NOMBRE:departamento {: nuevoDepartamento(departamento); :}
COMA CIFRA;
```

*La tercera funcionalidad que hemos implementado nos permite saber los trabajadores cuyo departamento o cargo no corresponda a un elemento ya declarado en la lista anterior. Para ello hemos tenido que implementar otra función y es la siguiente:

```
private boolean cargoOK(String cargo){
          if (!listaCargos.contains(cargos.trim())){
               System.out.println("Error: cargo " + cargo +
                                  "no definido. ");
               return false;
          }
          else {
               return true;
          }
     }
```

Igual que nos pasaba con la otra función que nos decía si el cargo estaba repetido o no, esta función no hace nada por si sola si no se llama desde la gramática. Para llamarla debemos hacer algo muy parecido:

```
PERSONA::=CIFRA PYC CONTRASENYA PYC NOMBRE PYC
NOMBRE:trabajador{: nuevoTrabajador(trabajador); :} PYC
NOMBRE:cargo {:cargoOK(cargo);:} PYC
NOMBRE:departamento {:departamentoOK(departamento);:}  PYC
CORREO PYC TELEFONO SALTO;
```

*La cuarta y última funcionalidad nos permite saber el número de trabajadores y para ello hemos creado otra función (igual que con cargos y departamentos) para añadir en un array los trabajadores no repetidos que haya en la empresa. Dicha función es:

```
private void nuevoTrabajador(String trabajador){
     if (listaTrabajadores.contains(trabajador)){
          System.out.println("Aviso: trabajador " +
                              trabajador + "  repetido.");
          }
          else{
               listaTrabajadores.add(trabajador.trim());
               trabajadores++;

          }
     }
```

Para saber el número de trabajadores e imprimirlo por pantalla tras finalizar el análisis he creado una variable estática que aumente cada vez que se añada un trabajador. Al final del análisis llamamos desde el main a dicha variable estática para que la imprima por pantalla.