Juan Enrique Antón Marazuela

Diego Jara López

PRÁCtica obligatoria: pROCESADORES DE LENGUAJE

Especificación léxica y sintáctica

# Introducción.

En esta memoria, los alumnos Juan Antón y Diego Jara de la asignatura Procesadores de Lenguajes perteneciente al grado de Ingeniería Informática, exponen una breve explicación sobre cómo ha sido el trabajo realizado en la especificación léxica y sintáctica, con el objetivo de diseñar e implementar un traductor de programas escritos en un lenguaje similar a PASCAL (lenguaje fuente), a otro similar a C (lenguaje final).

# Software.

Durante la realización de esta práctica, hemos utilizado las siguientes herramientas:

* JFlex: Mediante un archivo de especificación léxica (con extensión .flex) generamos un archivo Java, que simula un autómata reconocedor de determinadas cadenas. Una vez compilado este fichero, al ejecutarlo junto a una cadena de entrada (usualmente otro archivo) acepta o muestra un error, dependiendo de si ha reconocido alguna de las expresiones que fueron definidas en la especificación.
* Cup: Mediante un archivo de especificación sintáctica (con extensión .cup), generamos dos archivos Java: sym, el cual contiene todos los símbolos terminales de la gramática que queremos reconocer; y parser, el cual reconoce si la cadena de entrada puede reconocerse con una determinada producción, y de esa forma realizar la comprobación sintáctica del fichero.
* Eclipse Mars: Utilizado como soporte de entorno para la compilación, ejecución y estructura del programa, el cual se comporta como analizador léxico y sintáctico al mismo tiempo.

# Especificación léxica.

El primer paso, fue la creación de un archivo llamado “especificacionLexica.flex”, encargado de reconocer los lexemas especificados en el enunciado de la práctica, y producir los tokens correctos que servirán como cadena de entrada del analizador sintáctico.

Para ello, declaramos expresiones regulares que se ajustan a los requisitos del lenguaje, las cuales son identificadores, constantes numéricas (enteras y reales), constantes literales, comentarios y saltos de línea.

En el estado inicial <YYINITIAL>, además de declarar todo lo anterior, también añadimos expresiones regulares para las palabras reservadas del lenguaje y las sentencias de control de flujo.

Con este archivo creado, el siguiente paso fue crear el archivo Java que simulase ese autómata gracias a la herramienta JFlex (como hemos comentado en el apartado 2), generando el fichero “AnalizadorLexico.java”, el cual es compilado y ejecutado junto a un archivo de prueba para comprobar la correcta definición de los lexemas y la producción de errores para los que no estén bien definidos, como “23contador” refiriéndonos a un identificador.

Una vez comprobado el correcto funcionamiento del analizador léxico, sustituimos las sentencias asociadas a las expresiones regulares definidas, eliminando la impresión por pantalla del lexema reconocido (por ejemplo: System.out.println (“identifier”)), y añadiendo la devolución del símbolo reconocido que interpretará el analizador sintáctico (por ejemplo: return new Symbol (sym.identifier)).

# Especificación sintáctica.

Una vez realizada la especificación léxica, creamos un archivo con extensión .cup llamado “especificacionSintactica”, el cual se encarga de comprobar la correcta sintaxis del lenguaje gracias a la gramática definida en el enunciado.

En la zona de declaración de este archivo, se encuentran los símbolos terminales en primer lugar, los no terminales a continuación, y por último las preferencias entre los terminales, en el caso de que estos entren en conflicto.

Aparte, al final del fichero, se define la gramática mediante derivaciones de los no terminales hasta llegar a un estado final, que estará compuesto por un terminal.

Este archivo se le pasa al programa Cup mediante línea de comandos, para que genere los dos archivos comentados previamente, ‘sym.java’ y ‘parser.java’.

Una vez introducidos estos archivos en el proyecto, necesitamos modificar ‘parser.java’, ya que da errores de compilación en las producciones de los símbolos lambda. Este error se soluciona mediante la adición de un parámetro más al constructor de la clase Symbol, dicho parámetro es el mismo que se encuentra antes del objeto RESULT.

# Traducción dirigida por la sintaxis.

Para la traducción dirigida por la sintaxis, utilizamos el fichero “especificacionSintactica.cup” generado en la etapa anterior. El procedimiento seguido fue la introducción de acciones semánticas entre, y sobre todo, tras las sentencias gramaticales.

El primer paso fue identificar las sentencias gramaticales básicas, y a partir de ahí, generar un nueva cadena de texto que se traspasa al padre de esa sentencia ({: RESULT = new String (valor);:}), para que de esta manera él pueda generar una nueva cadena, hemos trabajado con esta forma hasta llegar a la sentencia gramatical más importante (la cual no tiene padre), “PRG”.

Hemos tenido que realizar excepciones con algunas cadenas como pueden ser:

* Las **constantes**: en el momento en el que identificamos una constante, la añadimos a un ArrayList, el cual será impreso en el fichero de salida en primer lugar.
* Las **funciones**: al igual que con las constantes, las hemos recogido en un ArrayList, que también es impreso en el fichero tras estas.
* **Return** de las funciones: Para identificar en que momento una función debe devolver su valor, ha sido necesaria la creación de dos variables auxiliares: un booleano llamado “funcion”, el cual se activa cuando entra en una sentencia gramatical de la misma; y un String llamado “nombreFuncion”, que como su propio nombre indica, almacena el nombre de la función que está reconociendo; gracias a estas dos variables, cuando llegamos a una asignación dentro de una función, podemos comprobar mediante una sentencia if-else si se trata de la asignación de valor de retorno o de cualquier otra.
* **Tabulaciones**: para tabular correctamente el código, hemos tenido que crear una variable y dos funciones auxiliares; la variable llamada “tabulaciones” lleva la cuenta de las tabulaciones que hay que realizar según en la parte del código en la que nos encontremos (se aumenta cada vez que encontramos un “{”, y disminuye en el caso de encontrar el contrario); la primera función llamada “calcularTabulaciones”, devuelve un String con el número de tabulaciones indicado en la variable previamente mencionada; mientras que la función llamada “calcularTabulacionesCorcheteFinal”, es la indicada de calcular cuantas tabulaciones hace falta para el símbolo “}”.
* **Incremento/Decremento** de los bucles for: es un caso similar al “return” de las funciones, mediante una variable booleana que se activa dependiendo de si hay un incremento o decremento, controlamos gracias a la sentencia if-else si hay que aumentar el valor de la variable contadora en el bucle, o en su defecto disminuirla.

En último lugar, hemos tenido que crear una clase extra (“EscribirFichero”), la cual se encarga de imprimir en el fichero, en el orden correspondiente, la traducción generada en el archivo cup.

Esta clase (“EscribirFichero”), implementa en primer lugar el cambio de extensión del archivo enviado, a un ‘.c’ mediante un método. Posteriormente, se encarga de recoger los diferentes parámetros traspasados por el analizador sintáctico (acciones semánticas), en los cuales se recogía el valor de diferentes Strings que contienen la traducción del lenguaje.

Ya con la traducción del lenguaje realizada por las diferentes acciones semánticas, esta clase se encarga de pintar estas colecciones de Strings (ArrayList) en el fichero con la extensión correcta ‘.c’ con el mismo nombre que el fichero de entrada.

Como se ha comentado antes, se imprimía el contenido de los diferentes ArrayList que contenían diferente información. Hay dos ArrayList, uno para funciones (que guarda su respectivo contenido) y luego otro para las constantes (estas son una de las excepciones que se han tenido que tomar en la traducción y que han sido comentadas antes).

# Programa principal.

Llamado “PracticaPL.java”, es el encargado de realizar la conexión del analizador léxico y analizador sintáctico, además de reconocer el número de casos de prueba que quieras pasar (introduciendo el nombre del archivo con extensión .pas).

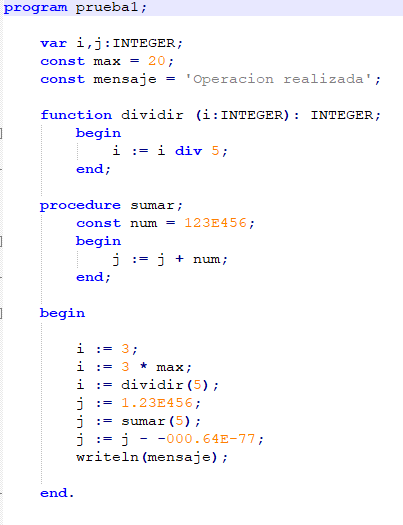
Dentro de esta clase, crearemos un nuevo analizador léxico, el cual será un argumento para el constructor del analizador sintáctico (parser.java).

Para finalizar, llamamos al método “parse” de la clase parser, el cual realiza el análisis sintáctico.

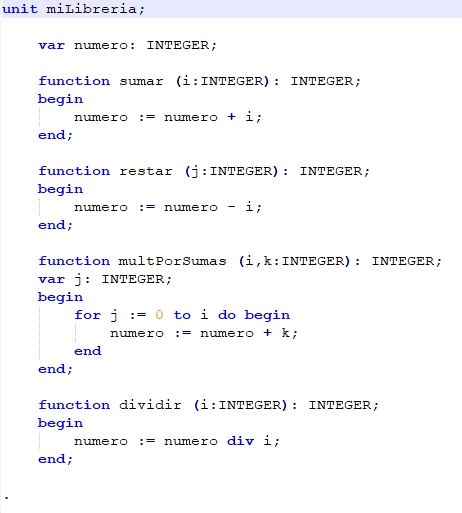
# Casos de prueba.

Hemos realizado ocho casos de prueba, cuatro correctos y cuatro incorrectos con las siguientes características:

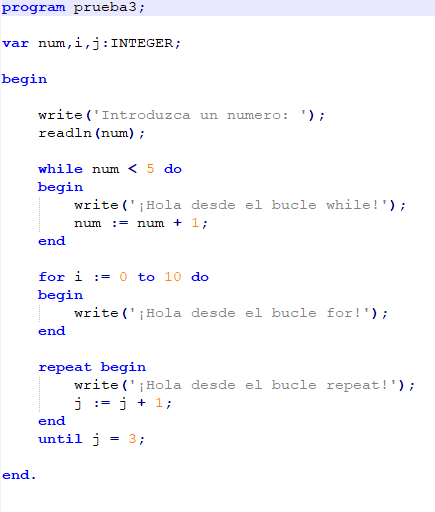
* pruebaCorrecta1.pas: simula un programa que realiza las operaciones de dividir y sumar mediante una función y un procedimiento.



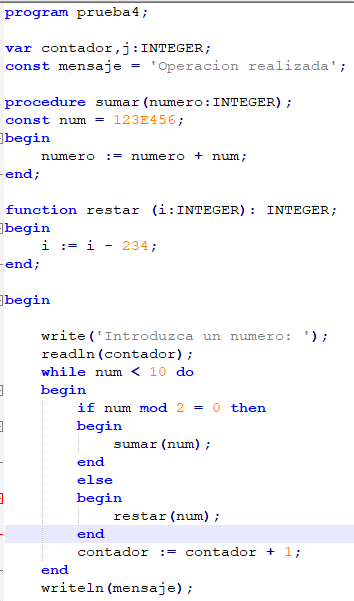
* pruebaCorrecta2.pas: simula el comportamiento de una librería para realizar operaciones aritméticas.



* pruebaCorrecta3.pas: simula un programa con distintos bucles, los cuales imprimen un determinado número de veces un mensaje por pantalla.



* pruebaCorrecta4.pas: simula un programa que realiza un determinado número de veces una operación u otra dependiendo de una condición.



Los casos de prueba incorrectos son iguales que los anteriores, introduciendo una modificación (anotada con un comentario en el código), para que estos ya no sean sintácticamente correctos:

* pruebaIncorrecta1.pas: omisión de un begin en la función “dividir”.
* pruebaIncorrecta2.pas: omisión de un punto y coma en el end de la función “restar”.
* pruebaIncorrecta3.pas: sustitución de una asignación por una comparación en el bucle for.
* pruebaIncorrecta4.pas: omisión del then en la sentencia condicional.