

# TRADUCCIÓN DIRIGIDA POR LA SINTAXIS

GRADO EN INGENIERÍA INFORMÁTICA

PROCESADORES DE LENGUAJES 18-19



AUTORES:

Miguel Taranilla de los Santos  
Sergio Hernández Domínguez  
Borja Martín Alonso

## **ÍNDICE**

---

<b>Introducción.....</b>	<b>1</b>
<b>Desarrollo de la práctica.....</b>	<b>2</b>
<b>Conclusión.....</b>	<b>16</b>

## Introducción

El objetivo de esta segunda parte de la práctica era el desarrollo de la traducción dirigida por la sintaxis, partiendo de los analizadores léxico y sintáctico ha sido posible completarlo para lograr la funcionalidad esperada de este procesador de lenguajes. Ante un mensaje de entrada escrito en el lenguaje Pascal, este procesador lo recibe, entiende y por último actúa en consecuencia para devolvernos ese mismo código en C.

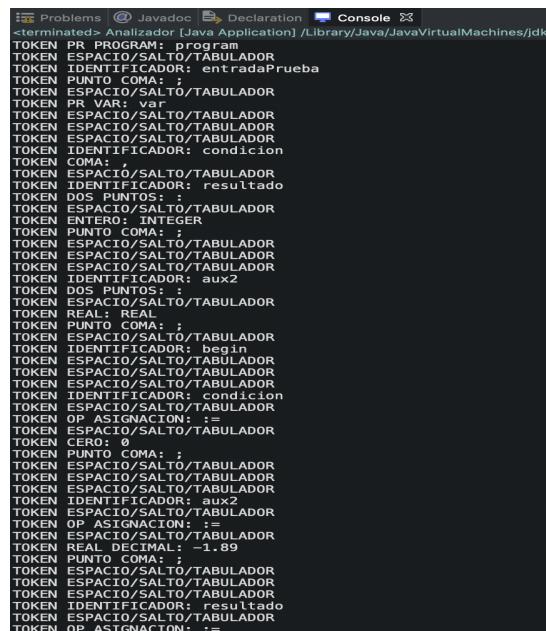
## Explicación del desarrollo de la práctica

### ESPECIFICACIÓN LÉXICA

En esta primera parte de la práctica, se implementa el analizador léxico. Para ello, se forma un .flex agrupando todos los patrones y macros relacionados con los tokens del lenguaje fuente, en este caso, PASCAL.

Además de agrupar patrones, expresiones regulares específicas y macros, se deben añadir un par de acciones simples con el fin de poder relacionarlo posteriormente con otras partes de la práctica.

Una vez realizado el código correspondiente al analizador léxico con todos los elementos del lenguaje PASCAL, se compila mediante consola para comprobar que no existe ningún error y en caso de ser una compilación exitosa, se pasa el código de prueba para ver que reconoce los tokens necesarios.



```
<terminated>- Analizador [Java Application] /Library/Java/JavaVirtualMachines/jdk
TOKEN PR_PROGRAM: program
TOKEN IDENTIFICADOR: entradaPrueba
TOKEN PUNTO COMA: ;
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN PR_VAR: var
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: condicion
TOKEN PUNTO COMA: ;
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: resultado
TOKEN DOS_PUNTOS: :
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ENTERO: INTEGER
TOKEN PUNTO COMA: ;
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: aux2
TOKEN DOS_PUNTOS: :
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN REAL: REAL
TOKEN PUNTO COMA: ;
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: begin
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: condicion
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN OP_ASIGNACION: :=
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: aux1
TOKEN PUNTO COMA: ;
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: aux2
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN OP_ASIGNACION: :=
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN REAL_DECIMAL: -1.89
TOKEN PUNTO COMA: ;
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: resultado
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN OP_ASIGNACION: :=
```

En esta imagen se puede ver como el analizador léxico reconoce todos los tokens del lenguaje fuente. Además, también incluimos el token espacio/tabulador/salto, para comprobar que todo funciona correctamente.

Imagen 1

A continuación, como ya hemos visto que funciona correctamente, introduciremos un fallo en nuestro código de prueba para poder ver cómo se comporta el analizador:

```

<terminated> Analizador [Java Application] /Library/Java/JavaVirtualMachines/jdk
TOKEN PR PROGRAM: program
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: entradaPrueba
TOKEN PUNTO COMA: ;
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN PR VAR: var
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: condicion
TOKEN COMA: ,
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: resultado
TOKEN DOS_PUNTOS: :
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ENTERO: INTEGER
TOKEN PUNTO COMA: ;
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: aux2
TOKEN DOS_PUNTOS: :
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN REAL: REAL
TOKEN PUNTO COMA: ;
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: begin
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN CERO: 0
TOKEN PUNTO COMA: ;
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: aux_2
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN OP_ASIGNACION: :=
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN CERO: 0
TOKEN PUNTO COMA: ;
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN IDENTIFICADOR: condicion
TOKEN OP_ASIGNACION: :=
TOKEN ESPACIO/SALTO/TABULADOR
TOKEN REAL_DECIMAL: -1.89
TOKEN PUNTO COMA: ;

```

Se puede apreciar en la imagen como al pasarle un token no reconocido en PASCAL (identificador que empieza por @), nos devuelve el error, mostrándonos la línea y columna en la que se produjo dicho error.

Imagen 2

En estos ejemplos anteriores podemos apreciar que en la primera parte del analizador lexico, no hemos preservado el formato ni la tabulación del código, además de que hemos reconocido los tokens ESPACIO/TABULADORES/SALTOS DE LINEA para posteriormente dependiendo de cada token, se muestre por pantalla con el formato correcto.

```

program entradaPrueba;
var
    condicion, resultado: INTEGER;
    aux2: REAL;
begin
    condicion := 0;
    aux2 := -1.89;
    resultado := condicion + aux2;
    writeln(resultado);
end.

```

Imagen 3

```

program entradaPrueba;
var
    condicion, resultado: INTEGER;
    aux2: REAL;
begin
    condicion := 0;
    aux2 := -1.89;
    resultado := condicion + aux2;
    writeln(resultado);
end.

```

Imagen 4

En estos dos ejemplos, se muestra como en la parte final del analizador lexico, se tiene ya en cuenta el formato de los códigos pasados y se ve como en caso de encontrar algún error lexico.

En caso de que se encuentre un error, como se puede apreciar en la imagen de la derecha, te marca el error donde se localiza (línea, columna) y la causa del error, en este caso token no reconocido @. Elegimos una descripción breve pero concisa de los errores debido a que debia

de devolver información valiosa para el programador sobre la causa del error y su localización.

## ESPECIFICACIÓN SINTÁCTICA

En cuanto a la parte de la especificación sintáctica, se ha creado un archivo .cup para añadir funcionalidad sintáctica a la parte de la especificación léxica implementada anteriormente.

En este fichero, se han añadido las reglas y producciones propias del lenguaje C con el fin de poder analizar sintácticamente y completar así esta primera parte del compilador.

Estas reglas muestran la estructura sintáctica que debe tener el programa que vayamos a analizar, regulando así su estructura y cada una de sus sentencias.

Este analizador sintáctico es generado mediante dos ficheros, sym.java y parser.java, ambos generados por la herramienta cup.

Finalmente, tras la implementación de cada especificación, ambas se deben de conectar con el fin de poder analizar al completo un código de prueba pasado por los usuarios. Una vez creada la conexión de ambas especificaciones mediante una clase Java que realice un trabajo sobre ambas clases y nos devuelva el resultado.

Para comenzar a crear el cup hemos necesitado una serie de variables terminales y no terminales para el correcto funcionamiento de este y que conseguirán crear el árbol sintáctico con dichos identificadores.

En la parte de los terminales incluimos identificadores aritméticos, caracteres especiales del lenguaje PASCAL a la hora de crear un programa (paréntesis, punto y coma, asignación, etc), identificadores de constantes tanto reales como enteras, y las palabras reservadas que contiene dicho lenguaje (begin, end, if, else, while, for, define, etc).

Por otro lado, tenemos los identificadores terminales son las reglas encargadas de realizar la sintaxis correcta de nuestro programa pasándole una serie de parámetros para controlar qué utilidad tendrá en el lenguaje C (si es una función, si está en la lista de variables del programa, qué tipo de operación se realiza en cada momento, etc).

Nos centramos específicamente en el fichero .cup describiendo las reglas más importantes a la hora de realizar la sintaxis de nuestro programa y obteniendo como resultado uno que sea sintácticamente correcto:

- PRG → aquí se describe cómo se identifica el programa y, al ir introduciendo el bloque de sentencias del éste, se generará un fichero con el nombre del programa.
- BLQ → su estructura comienza desde la palabra reservada BEGIN hasta END en cuyo cuerpo incluimos las declaraciones y sentencias pertinentes de nuestro programa para formar la sentencia de código que se visualizará.
- DCLIST → se va añadiendo declaraciones del programa principal a una lista en la que se irán guardando cada sentencia que aparezca en el cuerpo del programa.

- SENTLIST → comprobamos si la sentencia es una función, en tal caso, se crea una nueva función y se distingue si es una simple declaración de ésta o un control de flujo (ambas se añaden el bloque y las declaraciones de la función a la lista de sentencias).
- SENT → hay diferentes tipos de sentencias: asignación de un símbolo, llamada a un procedimiento y expresiones condicionales. Nos centramos más en esta última en la que si el nombre de la sentencia es un “if” recorrerá las declaraciones del bloque comprobando que sea una declaración de una función, en cuyo caso se añade a la lista de declaraciones, en caso contrario se añade a la lista del bloque (todas estas sentencias se visualizan en el bloque “then”). Si no se cumple la condición, todo el bloque de sentencias se incluye en la bloque del “else”. Podemos tener expresiones como “while-do” o “repeat-until” (en el que todas las declaraciones de la sentencia o de la lista de bloque se incluirá en él con la diferencia que en este caso se seguirá ejecutando dicho bloque hasta que no se cumpla la condición). Podemos tener la expresión “for”, en la que se van añadiendo las sentencias a la lista de declaraciones y a la lista de bloque para que después, a cada condición de la sentencia, se le vayan añadiendo los símbolos pertinentes (incrementando o decrementando el contador que recorre la expresión).
- LAMBDA → cuando en una regla aparece esta anotación, debemos dejar constancia que no se realizará nada, como un tipo de conclusión de esa regla.

El resto de los identificadores no terminales nos sirven para matizar qué componentes podemos encontrar en un programa:

1. definiciones de funciones, procedimientos, constantes o variables utilizadas
2. asignación de los diferentes símbolos en el programa.
3. introducción de parámetros que se le pueden pasar a funciones/procedimientos, incluyendo el valor que toma cada variable.
4. tipos definidos del lenguaje (real o entero).
5. símbolo de operaciones aritméticas, de asignación, lógicas, de signo.
6. palabra reservada de incremento o decremento en la expresión “for”.
7. anotación de factores de condición de las diferentes expresiones condicionales.

En caso de que existiese algún error de símbolos en la sintaxis a la hora de realizar el análisis del programa, éste nos indicará la fila y columna donde está localizado el fallo y nos recupera el error. En caso de ser un error no se recupere, el programa nos comentará que el error de la sintaxis no ha podido ser recuperado.

Una vez concluida la especificación del cup se debe ejecutar mediante el siguiente comando: “java -jar java\_cup.jar practicaObligatoria.cup”. Al ejecutarlo directamente se generan dos ficheros java denominados “parser” y “sym”:

- parser.java → analiza todas las reglas introducidas en el fichero cup para poder escribirlas en el lenguaje de programación Java de manera más entendible.
- sym.java → genera un número constante para cada parametro terminal del fichero cup y guarda el nombre de dichos terminales en un array de String.

## TRADUCCIÓN DIRIGIDA POR LA SINTAXIS

En esta tercera parte de la practica, se implementa la Traducción Dirigida por la Sintaxis, siendo esta una parte fundamental de la practica, ya que es donde se realiza realmente la traducción desde el lenguaje fuente (PASCAL) al lenguaje final (C).

Para ayudarnos en esta parte hemos partido primero de la construcción de varios arboles de traducción ascendentes, como el que le mostramos a continuación:

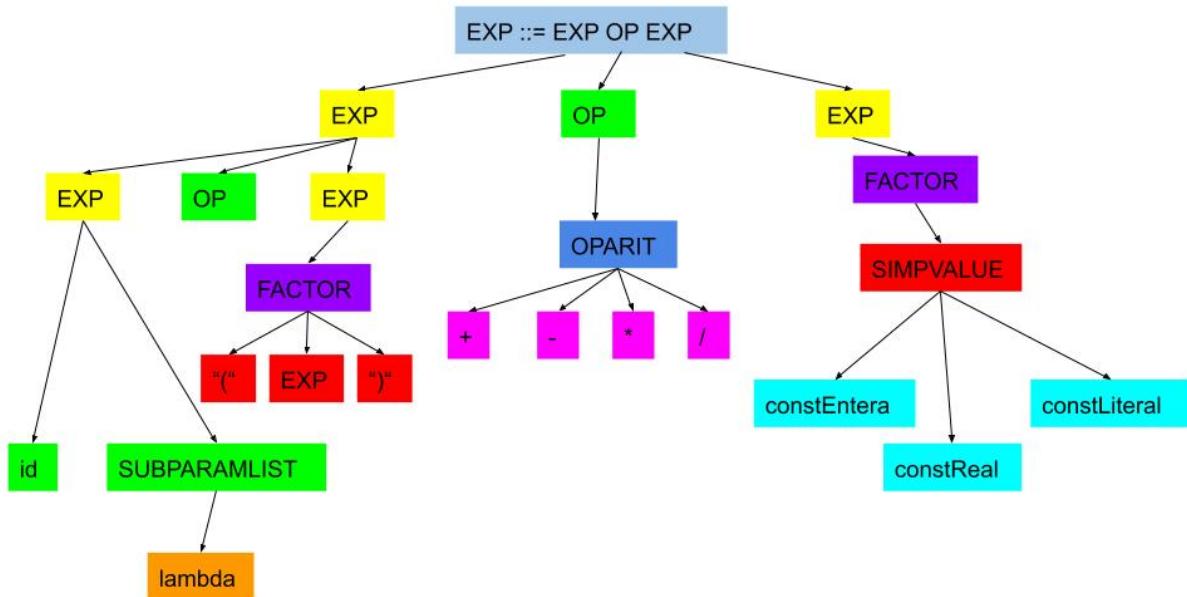


Imagen 5

En este árbol mostramos en forma de esquema el funcionamiento de una regla básica del lenguaje PASCAL como es la regla EXP.

Se puede apreciar como el árbol se forma mediante recursividad y va creciendo en función de las elecciones del programador a la hora de formar una expresión tan básica y sencilla como una expresión aritmética.

Como continuación de esta parte de la práctica, se ha incluido en el .cup, que ya teníamos formado de la parte anterior de la práctica (especificación sintáctica), las diferentes acciones semánticas. Apoyándonos en estas acciones semánticas, conseguiremos el comportamiento deseado para realizar el traductor de un lenguaje a otro.

Además, estas acciones semánticas, se ven apoyadas directamente en una serie de clases Java las cuales hacen la lógica y el funcionamiento de máquina de estados y de almacenamiento de

tipo pila necesario para formar los diferentes arboles de traducción y las relaciones entre los diferentes pares PASCAL-C.

A continuación, se muestran diferentes ejemplos de las acciones incluidas en el cup:

```

464 EXP ::= EXP:exp OP:op EXP:exp2 {:
465   Expresion e=new Expresion();
466   Simbolo s=new Simbolo();
467   s.esOperacionAritmetica=true;
468   s.operacion=op.operacion;
469   e.simbolos.addAll(exp.simbolos);
470   e.simbolos.add(s);
471   e.simbolos.addAll(exp2.simbolos);
472   RESULT=e;
473   :}
474   | FACTOR:factor {:
475     RESULT=exp;
476   :};
477
478 OP ::= OPARIT:oparit {:
479   RESULT=oparit;
480   :};
481
482 OPARIT ::= opMas {:
483   Simbolo s=new Simbolo();
484   s.esOperacionAritmetica=true;
485   s.tipoOperacionAritmetica="+";
486   RESULT=s;
487   :}
488   | opMenos {:
489   Simbolo s=new Simbolo();
490   s.esOperacionAritmetica=true;
491   s.tipoOperacionAritmetica="-";
492   RESULT=s;
493   :}
494   | opMultiplicacion {:
495   Simbolo s=new Simbolo();
496   s.esOperacionAritmetica=true;
497   s.tipoOperacionAritmetica="*";
498   RESULT=s;
499   :}
```

Imagen 6

Aquí podemos ver el ejemplo mostrado en el árbol de traducción de la regla EXP::=EXP OP EXP.

Primeramente, nos fijamos en la regla con cabecera EXP:

1. Se crea un objeto Expresión que es donde guardaremos todo el resultado de la regla y será devuelto.
2. Se crea además un tipo Símbolo que es donde se guardarán los símbolos que aparezcan en la expresión (símbolos de la regla OP)
3. Finalmente se añade al tipo lista de Símbolos que alberga el tipo Expresión las listas que se pueden encontrar en los subtipos recursivos exp y exp2 que forman la regla (línea código 468-471).

Por el lado de los símbolos de OP, podemos ver en la regla que son los símbolos correspondientes a OPARIT, los cuales son los relacionados con expresiones aritméticas sencillas (“+”, “-”, “\*”, “/”).

```

336     | estIf EXPCOND:expcond estThen BLQ:blq estElse BLQ:blq2 {:
337     Programa.contadorSentencias++;
338     Sentencia sentencia=new Sentencia();
339     sentencia.nombre="if";
340     for(Sentencia sAux: blq.declaraciones){
341         if(sAux.esDeclaracionFuncion){
342             sentencia.declaraciones.add(sAux);
343         }
344     }
345     for(Sentencia sAux: blq.bloque){
346         if(sAux.esDeclaracionFuncion){
347             sentencia.declaraciones.add(sAux);
348         }else{
349             sentencia.bloque.add(s);
350         }
351     }
352     for(Sentencia sAux: blq2.bloque){
353         if(sAux.esDeclaracionFuncion){
354             sentencia.declaraciones.add(sAux);
355         }else{
356             sentencia.bloqueElse.add(sAux);
357         }
358     }
359     sentencia.condicion.addAll(expcond.condicion);
360     sentencia.esControlFlujo=true;
361     sentencia.numeroSentencia=Programa.contadorSentencias;
362     RESULT=sentencia;
363   :}
364   | estWhile EXPCOND:expcond estDo BLQ:blq {:
365   Programa.contadorSentencias++;
366   Sentencia sentencia=new Sentencia();
367   sentencia.nombre="while";
368   for(Sentencia sAux: blq.declaraciones){
369       if(sAux.esDeclaracionFuncion){
370           sentencia.declaraciones.add(sAux);
371       }
372   }

```

A continuación, vamos a ver una parte de la regla cuya cabecera es SENT; en este caso la parte a comentar es el de las condiciones:

Los terminales con lo que se va a trabajar son estIf, estThen y estElse; vamos a analizarlo paso a paso.

1. Lo primero que se hace es aumentar el contador de sentencias del programa.
2. Creamos el objeto de la clase sentencia y lo rellenamos con el nombre que va a tener, en este caso al tratarse de un bloque if-then-else; será “if”.
3. Como se puede observar, a continuación lo que hay son tres bucles FOR , que son los encargado de ir recorriendo el array de declaraciones, bloque1 y bloque2 respectivamente; el tipo de datos de cada uno de los elemento del array es Sentencia.
4. Lo que se hace dentro de cada uno de los bucles for es ir mirando si por cada una de las sentencias contenidas dentro de este arrayList, es de tipo declaración función; en cuyo caso se irán añadiendo al array “sentencia.declaraciones” y en caso contrario se añade a otro array que contiene el resto de sentencias “sentencia.bloque”.

El resultado a devolver es ese objeto nuevo que hemos creado e ido rellenando.

```

405     | estFor identificador:id opAsignacion EXP:exp INC:inc EXP:exp2 doPR BLO:blo
406 Programa.contadorSentencias++;
407 Simbolo s=new Simbolo();
408 s.identificador=id;
409 s.esIdentificador=true;
410 Simbolo s2=new Simbolo();
411 s.asignacion=":=";
412 s.esAsignacion=true;
413 Sentencia sentencia =new Sentencia();
414 sentencia.nombre="for";
415 for(Sentencia sAux: blq.declaraciones){
416     if(sAux.esDeclaracionFuncion){
417         sentencia.declaraciones.add(sAux);
418     }
419 }
420 sentencia.condicionFor.add(s);
421 sentencia.condicionFor.add(s2);
422 sentencia.condicionFor.addAll(exp.condicion);
423 sentencia.condicionFor2.add(s);
424 sentencia.condicionFor2.add(inc.valor);
425 sentencia.condicionFor2.addAll(exp2.condicion);
426 for (Sentencia sAux : blq.bloque){
427     if (sAux.esDeclaracionFuncion){
428         sentencia.declaraciones.add(sAux);
429     }else {
430         sentencia.bloque.add(sAux);
431     }
432 }
433 sentencia.esControlFlujo=true;
434 sentencia.esCondicionFor=true;
435 sentencia.esCondicionFor2=true;
436 sentencia.identificadorFor=s;
437 sentencia.exp1=exp;
438 sentencia.exp2=exp2;
439 sentencia.numeroSentencia=Programa.contadorSentencias;
440 RESULT=sentencia;
441 ::;
442
443

```

Imagen 8.

Finalmente, como ultimo ejemplo de las acciones semánticas veremos el bucle FOR, dentro de la regla con cabecera SENT. Un ejemplo bastante completo donde podemos ver el funcionamiento de más extenso y general de lo explicado en esta tercera parte de la práctica. Primeramente, nos fijaremos en el cuerpo de la regla, donde podemos ver como empieza por el terminal estFor, el cual se corresponde con la palabra reservada del lenguaje “FOR”. Una vez entramos en la regla podemos ver el funcionamiento de las acciones que se realizan dentro:

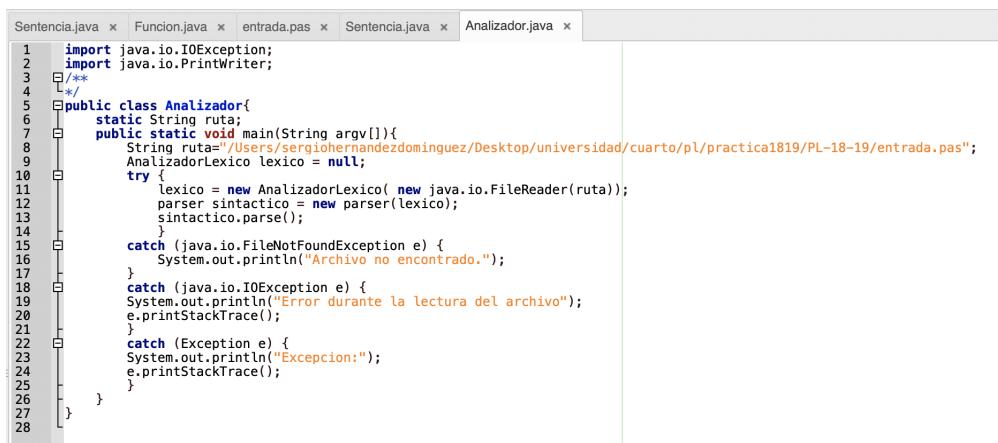
1. Se aumenta el número de sentencia de la clase Programa. Esto es básico debido a que luego en la clase Java que albergan a la mayoría (Programa, Función, etc), se va a tener en cuenta el número de sentencia por el cual vamos ejecutando, con el fin de controlar fallos y flujos de ejecución.
2. Se crean varios objetos Símbolo y se le da valor a los atributos identificador, esIdentificador, asignación, esAsignacion, etc.  
Con esto, podremos controlar los símbolos que se añaden en la condición del for para así incluirlas posteriormente.
3. Se crea un tipo Sentencia y se le da el nombre “for”, con el fin de que luego en la clase Java dependiendo del nombre, el comportamiento sea distinto.
4. Dentro ya del bloque del bucle, se va mirando cada sentencia que lo forma y se dividen dependiendo de la función que tenga la sentencia:
  - a. Si la sentencia es una declaración de función, se añade a la lista que alberga las declaraciones de funciones, ya que tiene un formato distinto.
  - b. Si la sentencia no es una declaración de función, se añade a la lista que contiene las demás sentencias, llamada bloque.
5. Además, se deben controlar las condiciones del FOR, ya que en un bucle FOR PASCAL tenemos solo dos condiciones y en cambio en un mismo bucle pero en el

lenguaje C, tenemos una condición más, la llamada incremento. Por eso se introducen en una lista llamadas condicionFor o condicionFor2 donde se alberga cada símbolo que forma parte de la condición además de los incrementos.

- Finalmente, se devuelve el tipo Sentencia en el que se alberga todo lo nombrado anteriormente.

\*Como aclaración, es posible que debido a mejoras en el cup y en las diferentes clases Java que forman este complejo analizador, se hayan modificado algunos atributos y nombres de operaciones posteriormente tras la realización de las capturas de pantalla, pero el comportamiento y el uso de ellos es el mismo.

Seguidamente, vamos a explicar algunos fragmentos de código de las diferentes clases Java que forman completamente la TDS.



```

1 import java.io.IOException;
2 import java.io.PrintWriter;
3 /**
4 */
5 public class Analizador{
6     static String ruta;
7     public static void main(String argv[]){
8         String ruta="/Users/sergioherandezdominguez/Desktop/universidad/cuarto/pl/practica1819/PL-18-19/entrada.pas";
9         AnalizadorLexico lexico = null;
10        try {
11            lexico = new AnalizadorLexico( new java.io.FileReader(ruta));
12            parser sintactico = new parser(lexico);
13            sintactico.parse();
14        }
15        catch (java.io.FileNotFoundException e) {
16            System.out.println("Archivo no encontrado.");
17        }
18        catch (java.io.IOException e) {
19            System.out.println("Error durante la lectura del archivo");
20            e.printStackTrace();
21        }
22        catch (Exception e) {
23            System.out.println("Excepcion:");
24            e.printStackTrace();
25        }
26    }
27 }
28

```

Imagen 9

En esta primera captura de pantalla, podemos ver una clase programada por nosotros para que relacione todas las clases y coja el fichero de prueba de donde esta almacenado.

Imagen 10



```

1 import java.util.ArrayList;
2
3 public class ControlFlujo {
4
5     public Sentencia cond;
6     public String nombre;
7     public ArrayList<Sentencia> bloque;
8     public ArrayList<Sentencia> bloqueElse;
9     public ArrayList<Sentencia> declaraciones;
10
11    public ControlFlujo() {
12        bloque = new ArrayList<>();
13        bloqueElse = new ArrayList<>();
14        declaraciones = new ArrayList<>();
15    }
16
17    public StringBuffer mostrarControlFlujo() {
18        StringBuffer sb = new StringBuffer();
19        if(nombre == "if") {
20            sb.append("(").append(" " + cond.mostrarSentencia().toString() + ")");
21            sb.append(" then\n");
22            sb.append("{\n");
23            for(Sentencia sent : bloque)
24                sb.append("\t" + sent.mostrarSentencia().toString() + "\r\n");
25            sb.append("}\r\n");
26            sb.append("else\n");
27            sb.append("{\r\n");
28            for(Sentencia sent : bloqueElse)
29                sb.append(sent.mostrarSentencia().toString() + "\r\n");
30            sb.append("}\r\n");
31        } else if(nombre == "for") {
32            sb.append("for ").append(cond.mostrarSentencia().toString());
33            sb.append("\r\n").append("{\r\n");
34            for(Sentencia sent : bloque)
35                sb.append(sent.mostrarSentencia().toString() + "\r\n");
36            sb.append("}\r\n");
37        } else if(nombre == "do") {
38            sb.append("do\n");
39            sb.append(" {\r\n");
40            for(Sentencia sent : bloque)
41                sb.append(sent.mostrarSentencia().toString() + "\r\n");
42            sb.append("}\r\n").append("until (" + cond.mostrarSentencia().toString() + ")");
43        }
44    }
45
46 }

```

A continuación podemos ver la clase ControlFlujo donde dependiendo del nombre, nos manda por cada una de las ramas del if donde se muestra un formato diferente y se realizan unas acciones diferentes.

```

1 import java.util.ArrayList;
2
3 public class Funcion {
4
5     public String nombre;
6     public String tipo;
7     public Sentencia param;
8     public ArrayList<Sentencia> declaraciones;
9     public ArrayList<Sentencia> bloque;
10    public boolean err;
11
12    public Funcion() {
13        declaraciones = new ArrayList<>();
14        bloque = new ArrayList<>();
15        param = new Sentencia();
16    }
17
18    public StringBuffer mostrarProcedimiento() {
19        StringBuffer sb = new StringBuffer();
20        if(!err) {
21            sb.append("\r\nvoid" + nombre + "(" + imprimirParams().toString() + "){\r\n");
22            if(declaraciones.isEmpty()) {
23                int nSent = declaraciones.get(0).numSentencia;
24                String simb = declaraciones.get(0).simbolos.get(0).mostrarSimbolo();
25                Simbolo simb2 = new Simbolo();
26                ArrayList<Sentencia> seguidas = new ArrayList<>();
27                for(Sentencia sent : declaraciones) {
28                    if(((sent.numSentencia == nSent) || (sent.numSentencia == (nSent+2))) && (sent.simbolos.get(0).mostrarSimbolo().equals(simb))) {
29                        seguidas.add(sent);
30                        nSent = sent.numSentencia;
31                    } else {
32                        Sentencia sentAux = new Sentencia();
33                        simb2.tipo = true;
34                        if(seguidas.get(0).simbolos.get(0).mostrarSimbolo().equals("int"))
35                            simb2.t = "int";
36                        else
37                            simb2.t = "float";
38                        for(Sentencia sent1 : seguidas) {
39                            for(Simbolo s : sent1.simbolos) {
40                                if((!s.tipo) && (!s.puntoYComa))
41                                    sentAux.simbolos.add(s);
42                            }
43                            sentAux.simbolos.add(simb2);
44                            sentAux.declaracionFunc = true;
45                            sb.append(sentAux.mostrarSentencia());
46                            seguidas.clear();
47                            seguidas.add(sent);
48                            nSent = sent.numSentencia;
49                            simb = sent.simbolos.get(0).mostrarSimbolo();
50                        }
51                    }
52                }
53                Sentencia sent2 = new Sentencia();
54                simb2.tipo = true;
55                if(seguidas.get(0).simbolos.get(0).mostrarSimbolo().equals("int"))

```

Imagen 11

Seguidamente nos encontramos con la clase Función, una de las principales clases de nuestro analizador. Esta se encuentra formada principalmente por dos ArrayList de tipo Sentencia, los cuales albergan dos tipos de sentencias:

1. Sentencias de tipo declaración de función (lista declaraciones)
2. Sentencias de tipo básico (lista bloque)

En esta imagen, se ve además el primer método de la clase llamado mostrarProcedimiento, en el cual se convierte un procedimiento de lenguaje PASCAL a una función del lenguaje C (lo que más llama la atención es que debido a que es un procedimiento, el parámetro de devolución en el lenguaje C sera “void”).

```

1 import java.util.ArrayList;
2
3 public class Expresion {
4
5     public ArrayList<Simbolo> simbolos;
6     public boolean error;
7
8     public Expresion() {
9         simbolos = new ArrayList<>();
10    }
11
12     public StringBuffer mostrarExpresion() {
13         StringBuffer sb = new StringBuffer();
14         for(Simbolo simbolo : simbolos)
15             sb.append(simbolo.mostrarSimbolo() + " ");
16     }
17 }
18
19 }
```

Imagen 12

Podemos ver aquí la clase Expresión, una clase muy sencilla donde se albergan los diferentes símbolos que forman una expresión en un ArrayList de tipo Símbolo, llamado simbolos. Seguidamente, se muestra el único método que forma la clase que es la de mostrarExpresion, el cual devuelve los Símbolos que forman parte de la lista “símbolos”.

```

1 import java.io.*;
2 import java.util.ArrayList;
3
4 public class Programa {
5
6     public static String nombreFunc;
7     public static int contSents;
8     public String nombre;
9     public ArrayList<Sentencia> declaraciones;
10    public ArrayList<Sentencia> variables;
11    public ArrayList<Sentencia> funciones;
12
13    public Programa() {
14        nombre = "PRUEBA PL: ";
15        declaraciones = new ArrayList<>();
16        variables = new ArrayList<>();
17        funciones = new ArrayList<>();
18    }
19
20    public StringBuffer imprimir() {
21        StringBuffer sb = new StringBuffer();
22        Sentencia principal = new Sentencia();
23        for(Sentencia f : funciones) {
24            if(f.nombre != null) {
25                if(f.nombre.equals("main"))
26                    principal = f;
27            }
28        }
29        if(!principal.declaraciones.isEmpty()) {
30            int nSent = principal.declaraciones.get(0).numSentencia;
31            String simb = principal.declaraciones.get(0).simbolos.get(0).mostrarSimbolo();
32            Simbolo simb2 = new Simbolo();
33            ArrayList<Sentencia> seguidas = new ArrayList<>();
34            for(Sentencia sent : principal.declaraciones) {
35                if((sent.numSentencia == nSent) || (sent.numSentencia == (nSent+2)) && (sent.simbolos.get(0).mostrarSimbolo().equals(simb))) {
36                    seguidas.add(sent);
37                    nSent = sent.numSentencia;
38                } else {
39                    Sentencia sentAux = new Sentencia();
40                    simb2.tipo = true;
41                    if(seguidas.get(0).simbolos.get(0).mostrarSimbolo().equals("int"))
42                        simb2 = "int";
43                    else
44                        simb2.t = "float";
45                    for(Sentencia sent1 : seguidas) {
46                        for(Simbolo s : sent1.simbolos) {
47                            if((s.tipo) && (!s.puntoYComa))
48                                sentAux.simbolos.add(s);
49                        }
50                    }
51                    sentAux.simbolos.add(simb2);
52                    sentAux.declaracionFunc = true;
53                    sb.append(sentAux.mostrarSentencia());
54                    seguidas.clear();
55                    seguidas.add(sent);
56                }
57            }
58        }
59    }
60 }
```

```
1  import java.util.ArrayList;
2
3  public class Simbolo {
4
5      public boolean identificador;
6      public String id;
7
8      public boolean constante;
9      public String cte;
10
11     public boolean tipo;
12     public String t;
13
14     public boolean opAritmetica;
15     public String op;
16
17     public boolean parentesis;
18     public String p;
19
20     public boolean coma;
21     public String c;
22
23     public boolean llave;
24     public String key;
25
26     public boolean operacionC;
27     public String opC;
28
29     public boolean operacionL;
30     public String opL;
31
32     public boolean negacion;
33     public String neg;
34
35     public boolean puntoYComa;
36     public String pyc;
37
38     public boolean retorno;
39     public String ret;
40
41     public boolean asignacion;
42     public String asig;
43
44     public boolean definicion;
45     public String def;
```

Imagen 13

En esta clase, a pesar de ser una clase extensa, es muy sencilla, ya que se encuentra formada en su mayoría por atributos con los cuales distinguimos los diferentes tipos de simbolos que forman el lenguaje. Además, como curiosidad, siempre es un par de atributos, uno boolean y uno String.

El atributo de tipo boolean nos servirá para controlar el tipo de símbolo que es y almacenarlo de manera correcta y el atributo de tipo String nos servirá para poder almacenar el valor real (léxico) del símbolo.

A continuación, se muestran las dos clases más complejas y las cuales controlan nuestro analizador.

```
1 import java.util.ArrayList;
2
3 public class Sentencia {
4
5     public boolean procSinParentesis;
6     public boolean error;
7     public boolean declaracionFunc;
8     public boolean declaracionCte;
9     public boolean devuelto;
10    public boolean paramFunc;
11    public boolean asign;
12    public boolean noTab;
13    public boolean controlFlujo;
14    public boolean esCondFor1;
15    public boolean esCondFor2;
16    public boolean esCondicion;
17
18    public static String nombreFunc;
19    public static int contTabs = 1;
20    public String nombre;
21    public int numSentencia;
22
23    public ArrayList<Simbolo> simbolos;
24    public ArrayList<Simbolo> condicion;
25    public ArrayList<Simbolo> condFor1;
26    public ArrayList<Simbolo> condFor2;
27    public ArrayList<Sentencia> bloque;
28    public ArrayList<Sentencia> bloqueElse;
29    public ArrayList<Sentencia> declaraciones;
30
31    public Simbolo idFor;
32    public Expresion e1;
33    public Expresion e2;
34
35    public Sentencia() {
36        simbolos = new ArrayList<>();
37        condicion = new ArrayList<>();
38        bloque = new ArrayList<>();
39        declaraciones = new ArrayList<>();
40        bloqueElse = new ArrayList<>();
41        condFor1 = new ArrayList<>();
42        condFor2 = new ArrayList<>();
43        e1 = new Expresion();
44        e2 = new Expresion();
45    }
46
47    public StringBuffer mostrarSentencia() {
48        StringBuffer sb = new StringBuffer();
49        if(declaracionCte) {
50            for(Simbolo s : simbolos) {
51                if(!s.getDefinicion) {
52                    if(s.getIdentificador)
53                        sb.append("\t" + s.mostrarSimbolo().toString() + ";");
54                else
55                    sb.append(s.mostrarSimbolo().toString() + ";");
56            }
57        }
58    }
59}
```

```
1 import java.io.*;
2 import java.util.ArrayList;
3
4 public class Programa {
5
6     public static String nombreFunc;
7     public static int contSents;
8     public String nombre;
9     public ArrayList<Sentencia> declaraciones;
10    public ArrayList<Sentencia> variables;
11    public ArrayList<Sentencia> funciones;
12
13    public Programa() {
14        nombre = "PRUEBA PL: ";
15        declaraciones = new ArrayList<>();
16        variables = new ArrayList<>();
17        funciones = new ArrayList<>();
18    }
19
20    public StringBuffer imprimir() {
21        StringBuffer sb = new StringBuffer();
22        Sentencia principal = new Sentencia();
23        for(Sentencia f : funciones) {
24            if(f.nombre != null) {
25                if(f.nombre.equals("main"))
26                    principal = f;
27            }
28        }
29        if(!principal.declaraciones.isEmpty()) {
30            int nSent = principal.declaraciones.get(0).numSentencia;
31            String simb = principal.declaraciones.get(0).mostrarSimbolo();
32            Simbolo simb2 = new Simbolo();
33            ArrayList<Sentencia> seguidas = new ArrayList<>();
34            for(Sentencia sent : principal.declaraciones) {
35                if((sent.numSentencia == nSent) || (sent.numSentencia == (nSent+2))) && (sent.simbolos.get(0).mostrarSimbolo().equals(simb))) {
36                    seguidas.add(sent);
37                    nSent = sent.numSentencia;
38                } else {
39                    Sentencia sentAux = new Sentencia();
40                    simb2.tipo = true;
41                    if(seguidas.get(0).simbolos.get(0).mostrarSimbolo().equals("int"))
42                        simb2.t = "int";
43                    else
44                        simb2.t = "float";
45                    for(Sentencia sent1 : seguidas) {
46                        for(Simbolo s : sent1.simbolos) {
47                            if(!s.tipo) && (!s.puntoYComa)
48                                sentAux.simbolos.add(s);
49                        }
50                    }
51                    sentAux.simbolos.add(simb2);
52                    sentAux.declaracionFunc = true;
53                    sb.append(sentAux.mostrarSentencia());
54                    seguidas.clear();
55                    seguidas.add(sent);
56                }
57            }
58        }
59    }
60}
```

## Imagen 14

Ambas clases serán las encargadas de controlar cada sentencia del código (clase Sentencia) y la encargada de mostrar el código correctamente en un fichero generado en una ruta especificada (clase Programa).

Además de las clases mostradas aquí, se encuentran las clases AnalizadorLexico.java, sym.java y parser.java. Estas clases son generadas con el software proporcionado por la universidad, jflex y cup. Por lo tanto, de estas clases no se ha proporcionado información relativa a estas.

Finalmente, a la hora de intentar ejecutar el analizador y ver su correcto funcionamiento, hemos visto que no ejecutaba de manera correcta y nos daban errores sintácticos, los cuales no aparecían en la parte de especificación sintáctica y mucho más extraño era que esos errores no se encontraban en el fichero de prueba que se le pasa.

## **Conclusión**

Gracias a esta práctica hemos podido profundizar mucho y de manera correcta en el mundo de los analizadores de lenguajes y los traductores. Además, nos ha servido para aprender y aclarar conceptos, los cuales luego hemos podido plasmar en la parte teórica de esta asignatura.

Finalmente, queremos agradecerle al profesor Pedro Paredes por su atención en las diferentes tutorías que hemos acudido para aclarar dudas y errores de esta práctica.