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**

**Introducción……………………………………………………………………1**

**Desarrollo de la práctica…………………………………………………2**

**Conclusión……………………………………………………………………16**

**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 a 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.

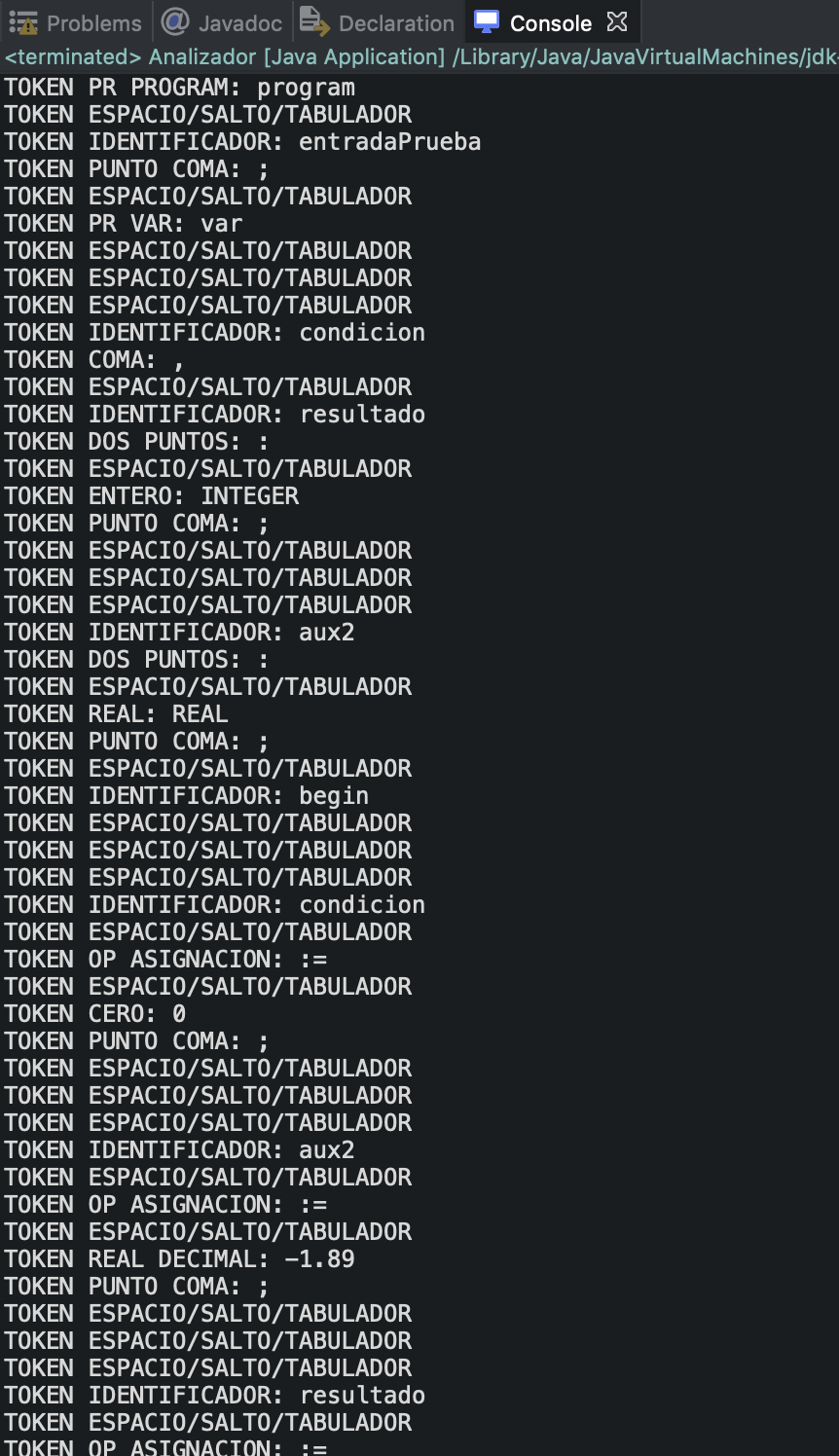
**Explicación del desarrollo de la práctica**

**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.

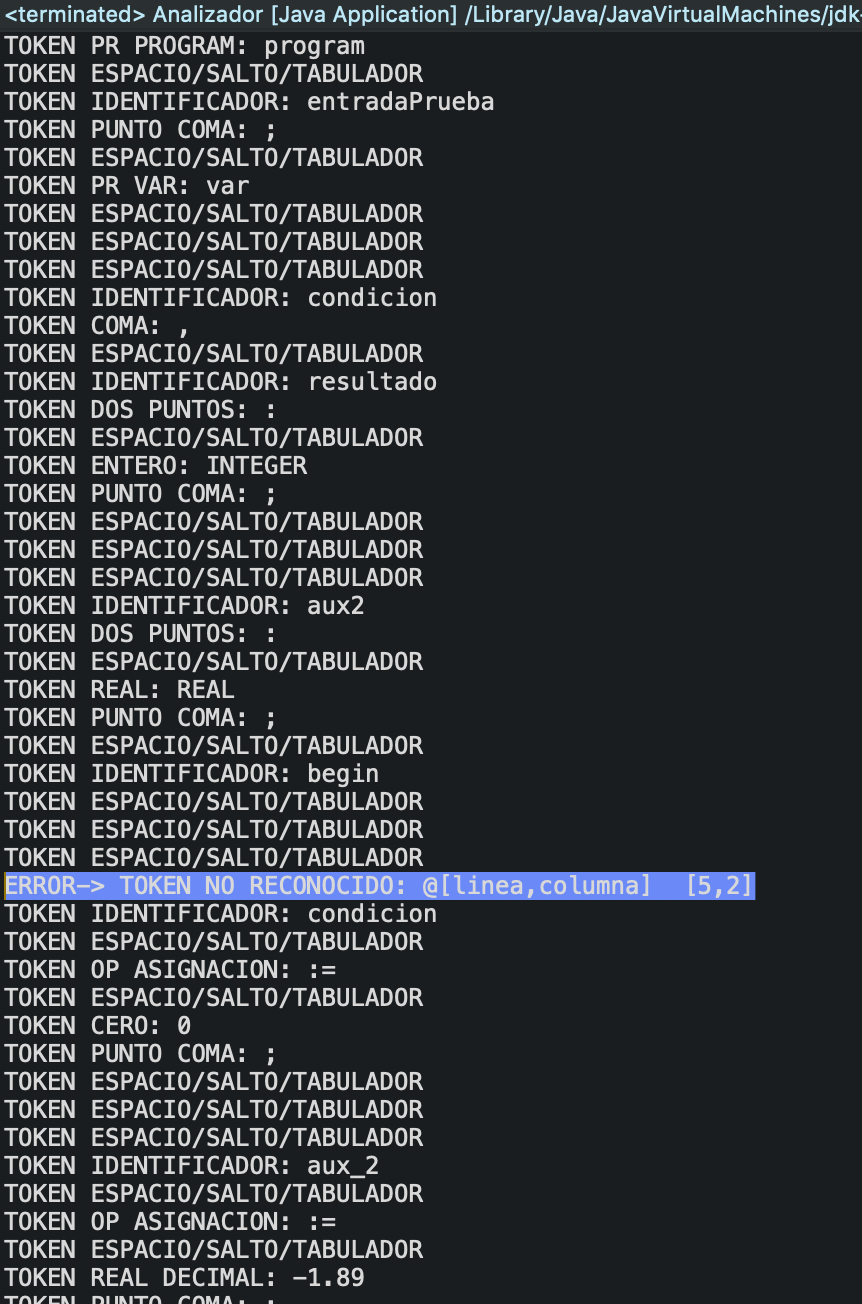
Una vez realizado el código correspondiente al analizador léxico con todos los elementos del lenguaje PASCAL, se compila mediante consola para comprobar que no existe ningún error y en caso de ser una compilación exitosa, se pasa el código de prueba para ver que reconoce los tokens necesarios.



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:



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

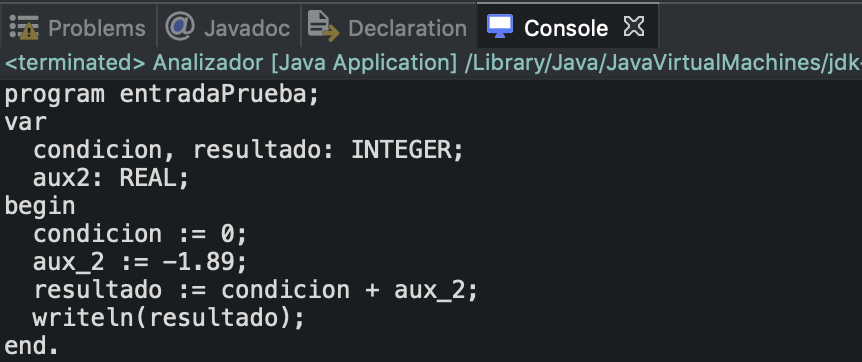
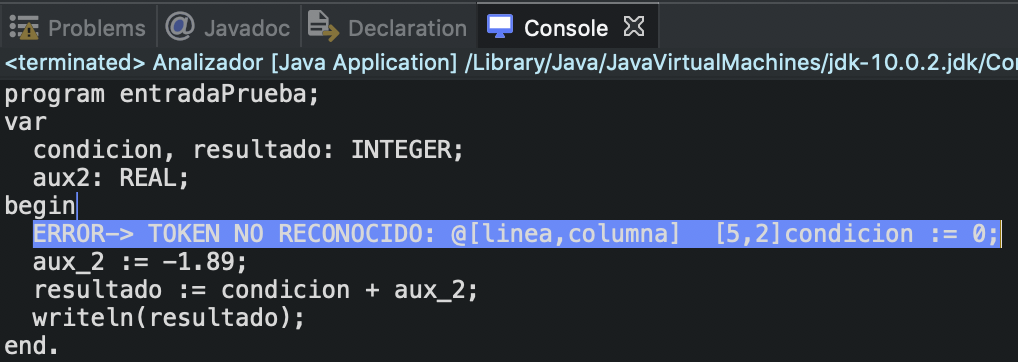
En estos ejemplos anteriores podemos apreciar que en la primera parte del analizador lexico, no hemos preservado el formato ni la tabulación del código, además de que hemos reconocido los tokens ESPACIO/TABULADORES/SALTOS DE LINEA para posteriormente dependiendo de cada token, se muestre por pantalla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

Imagen 3 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 esté 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 así́ esta primera parte del compilador.

Estas reglas muestran la estructura sintáctica que debe tener el programa que vayamos a analizar, regulando así́ 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 que contiene texto, mapa

Descripción generada automáticamente

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.

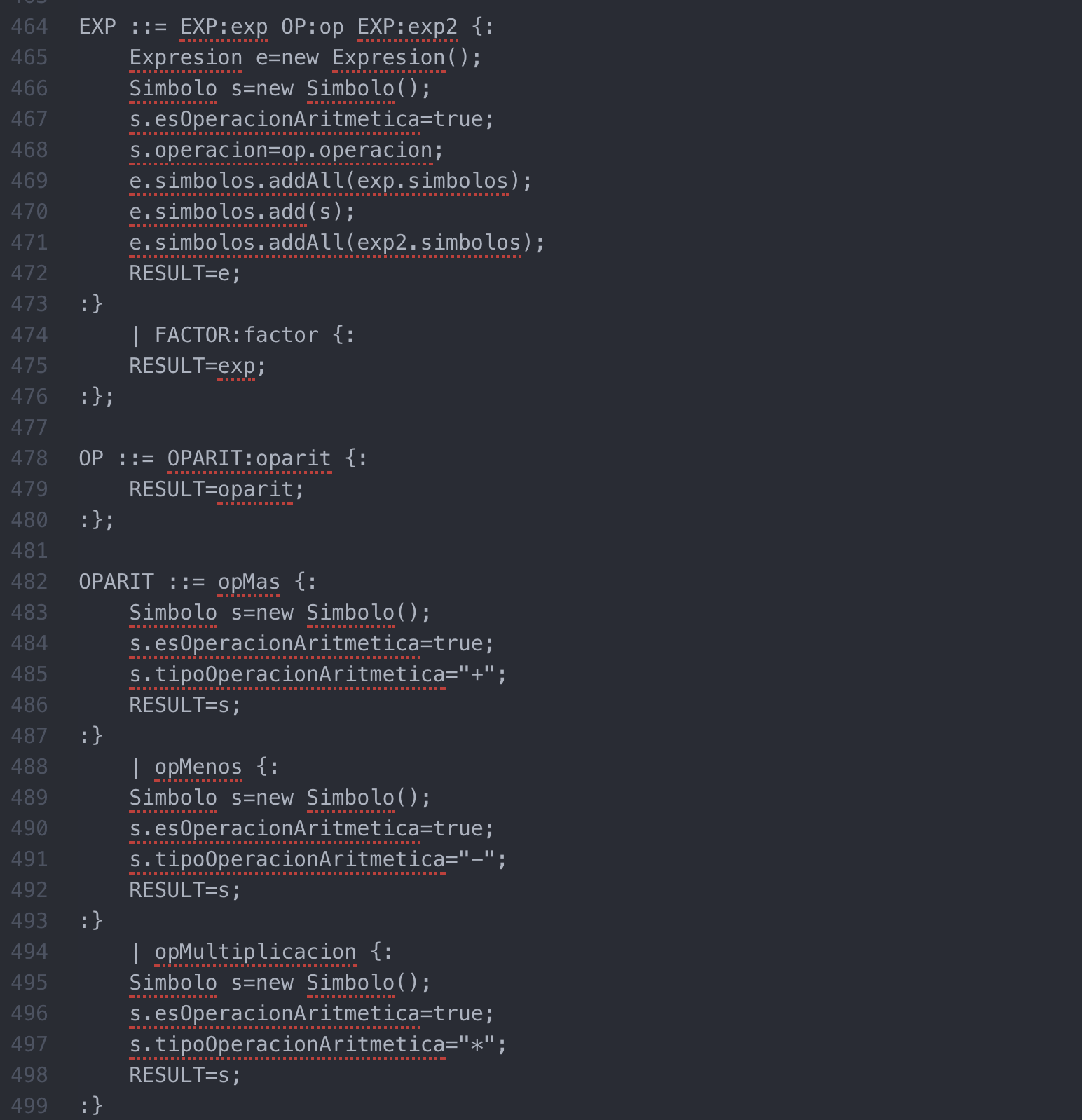
A continuación, se muestran diferentes ejemplos de las acciones incluidas en el cup:

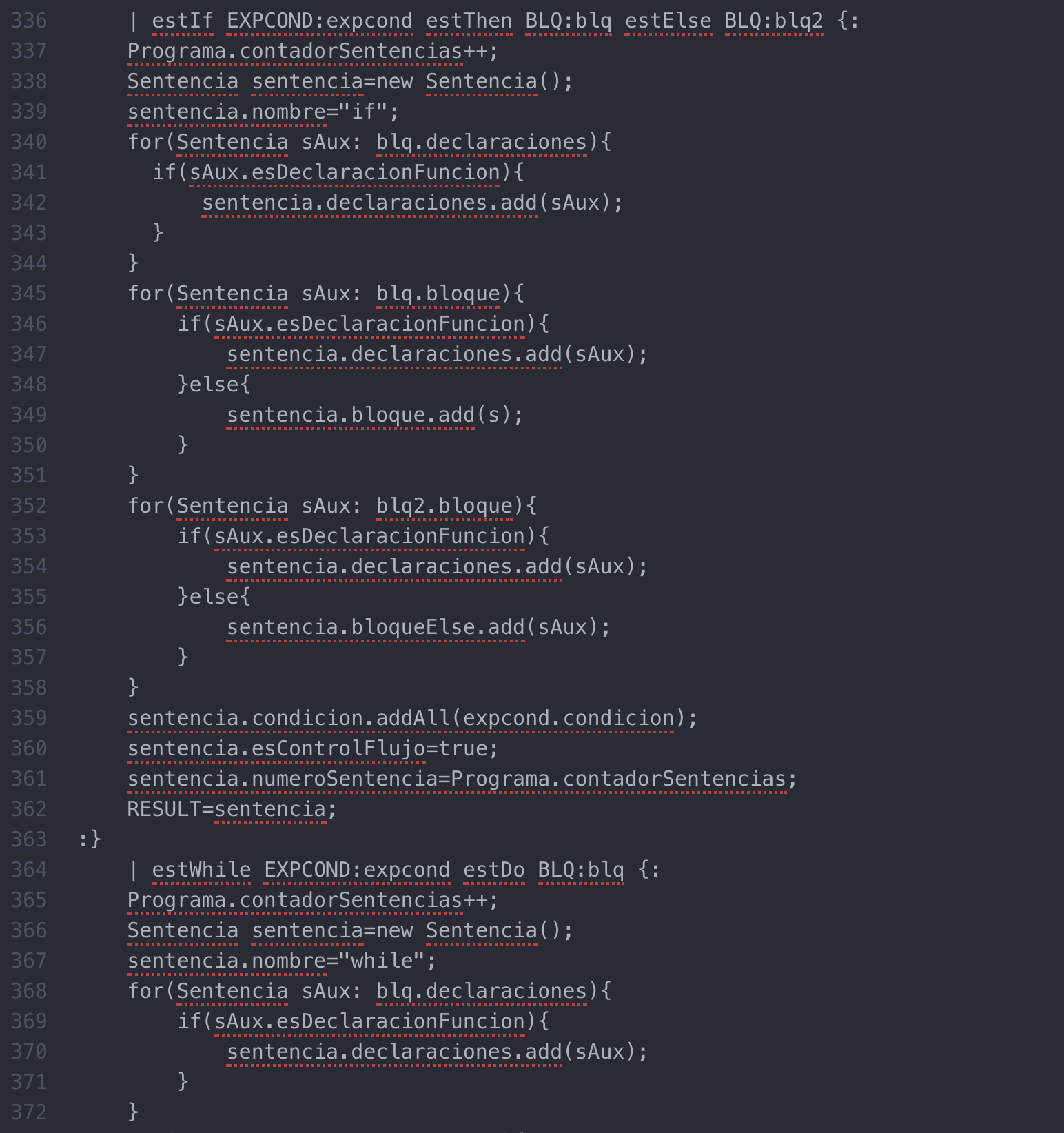
Imagen 6

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 guardaran 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).

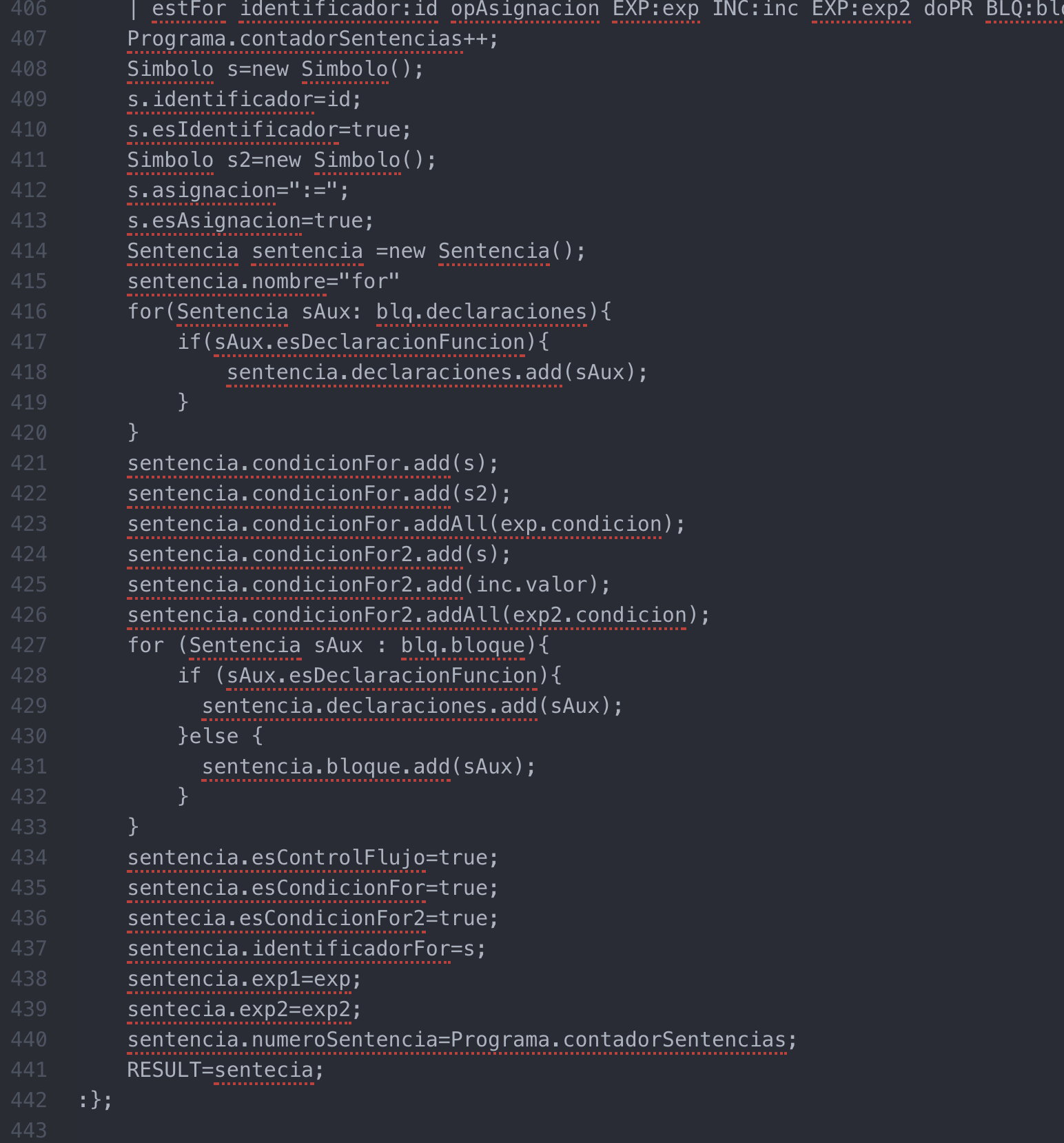
Por el lado de los símbolos de OP, podemos ver en la regla que son los símbolos correspondientes a OPARIT, los cuales son los relacionados con expresiones aritméticas sencillas (“+”, “-”, “\*”, “/”).

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.

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.

1. 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.
2. 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:
   1. 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.
   2. Si la sentencia no es una declaración de función, se añade a la lista que contiene las demás sentencias, llamada bloque.
3. 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.
4. 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 esté 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.

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

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.

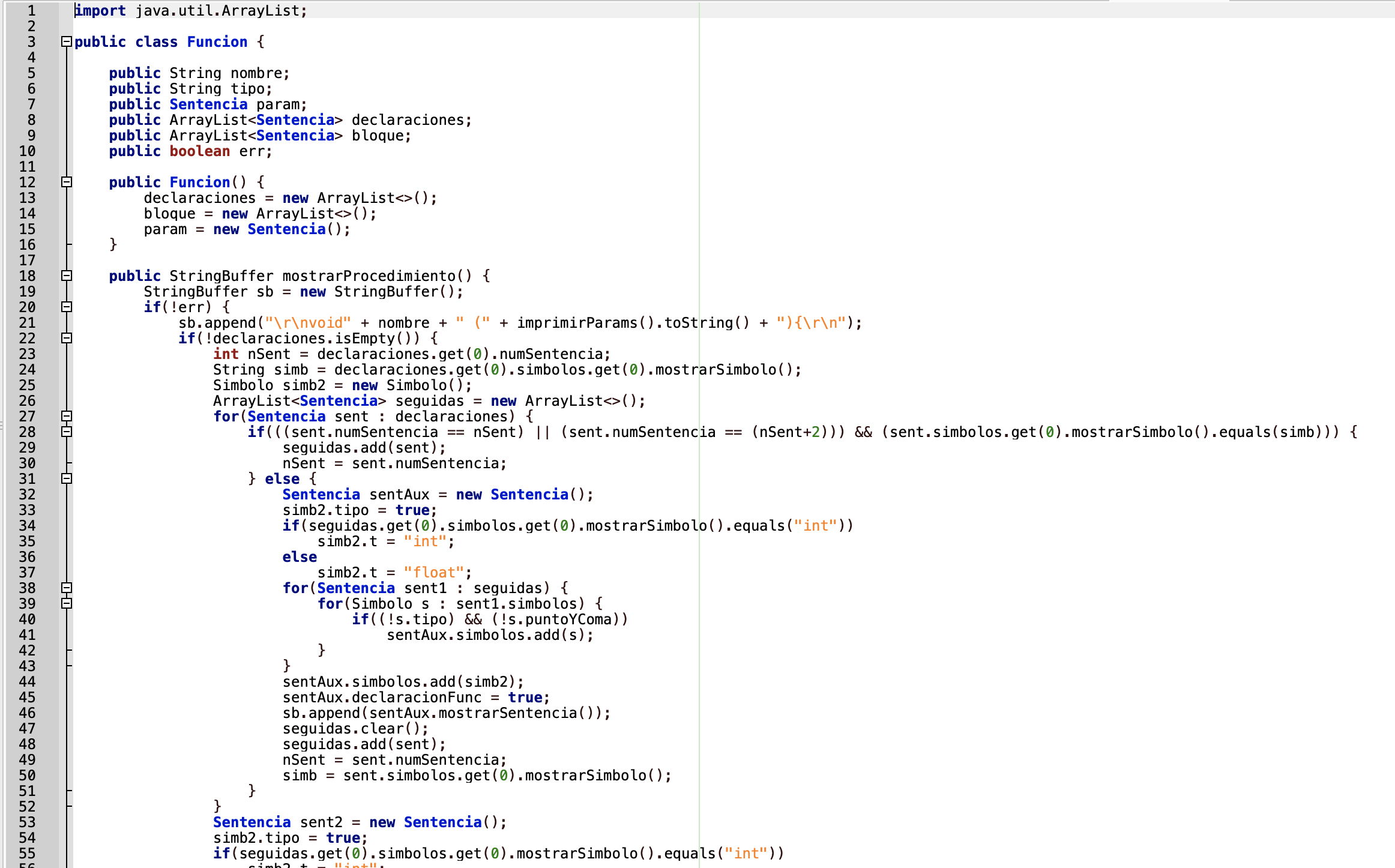
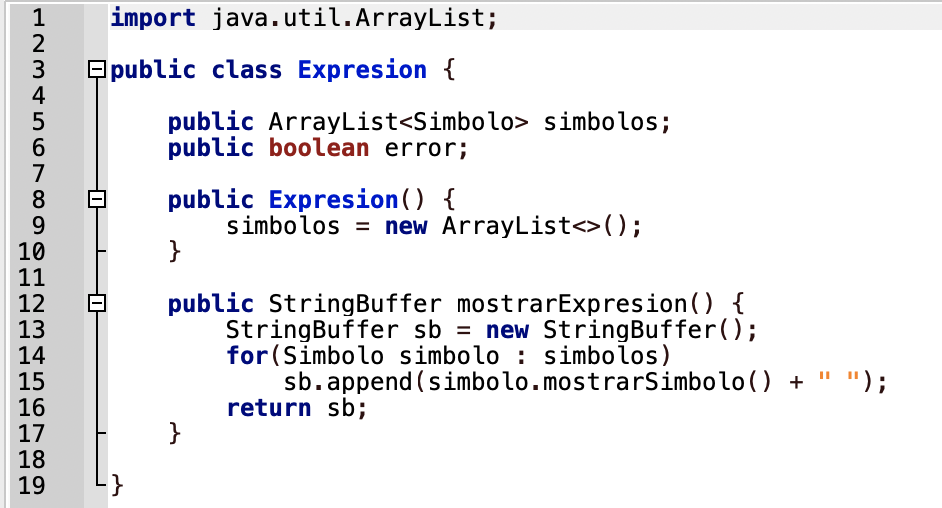


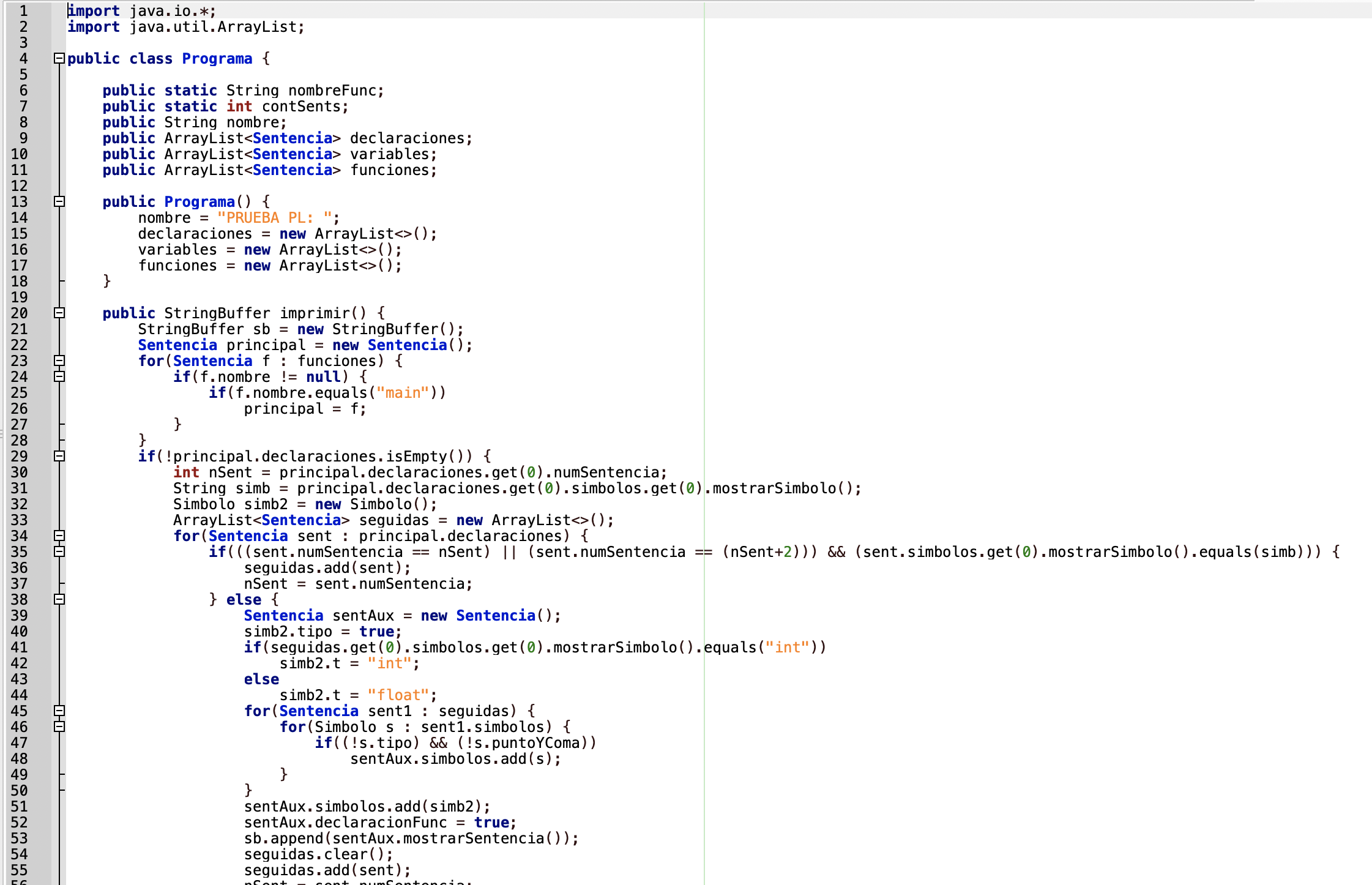
Imagen 11

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”).

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 símbolos. 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”.

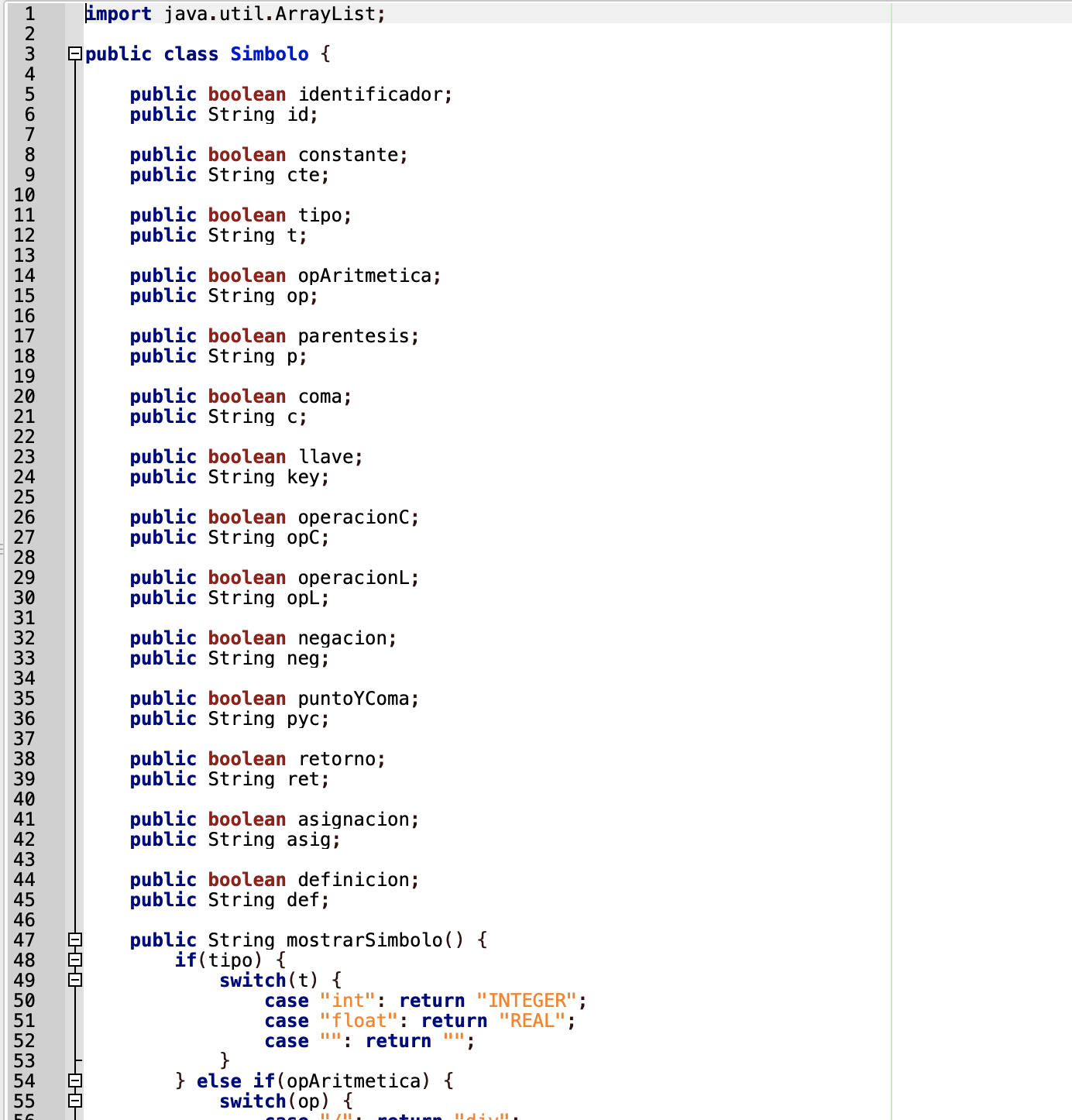
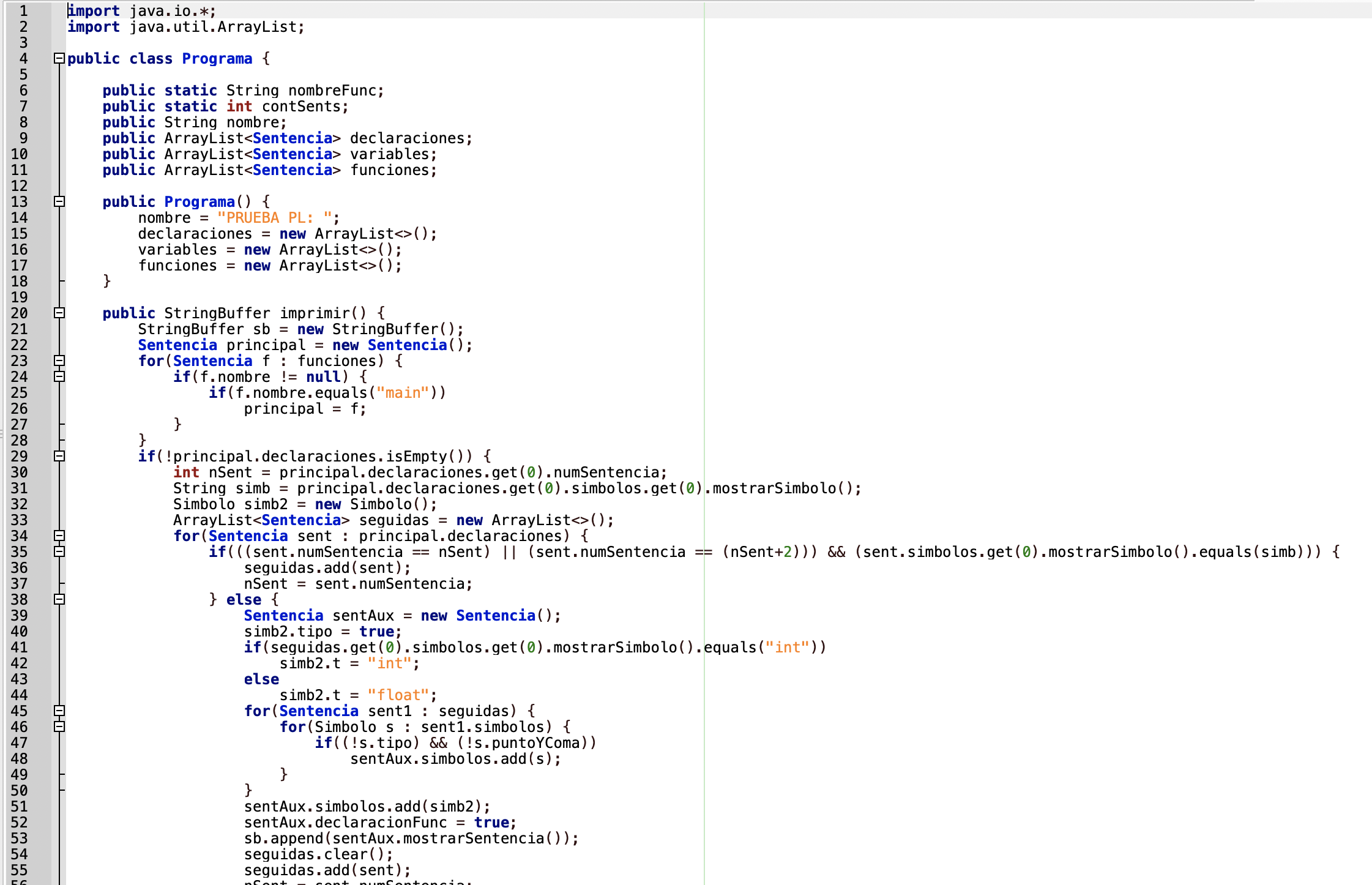


Imagen 13

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.

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 practica 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 practica.