1. ***Introducción:***

En este documento vamos a describir el proceso para desarrollar un procesador del lenguaje JavaScript. Este procesador contiene tanto analizador léxico, sintáctico y semántico, como tablas de símbolos, gestión de errores y los tokens correspondientes. Las especificaciones de nuestro grupo, **el grupo 9**, son las siguientes:

 Tipos de Datos (opcional): **Objetos**

 Sentencias: **Sentencia de selección múltiple (switch-case)**

 Operadores especiales: **Asignación con operación (op=)**

 Técnicas de Análisis Sintáctico: **Ascendente LR**

En nuestro caso, los objetos no han sido implementados puesto que eran opcionales. En cambio, sí que hemos incluido operadores opcionales que se especificarán más adelante.

Esta práctica ha sido implementada en el lenguaje de programación java. Además se ha utilizado la herramienta Anagra para la generación de las tablas de GoTo y acción del analizador sintáctico. Se explicará cómo funciona cada uno de los analizadores así como la gestión de las tablas de símbolos y se añadirá toda la información necesaria para poder comprender como trabaja este procesador.

1. ***Analizador Léxico:***

Nuestro programa toma un fichero JavaScript y lo procesa línea a línea y carácter a carácter, simulando el comportamiento del autómata y generando los tokens que va encontrando, ignorando elementos innecesarios y que no generan tokens como, por ejemplo, los comentarios. Cada vez que se genera un token identificador se inserta en la tabla de símbolos correspondiente y en caso de encontrarse una declaración de función, se crea una tabla de símbolos nueva.

1. **Tokens:**

Los tokens que puede procesar nuestro analizador son los siguientes:

|  |  |
| --- | --- |
| **TOKENS** | **DESCRIPCIÓN** |
| **<Cadena, cadena>** | Cadena con su cadena correspondiente |
| **<Entero, valor>** | Entero con su valor correspondiente |
| **<Id, posTS>** | Identificador con su posición en la TS |
| **<OpAritm, codigo>** | Operador aritmético con su código correspondiente  (códigos especificados después de la tabla) |
| **<OpAsig, código>** | Operador asignación con su código correspondiente  (códigos especificados después de la tabla) |
| **<OpLog, código>** | Operador lógico con su código correspondiente  (códigos especificados después de la tabla) |
| **<OpRel, código>** | Operador relacional con su código correspondiente  (códigos especificados después de la tabla) |
| **<PalRes, lexema>** | Palabra reservada |
| **< ( , - >** | Abre paréntesis |
| **< ) , - >** | Cierra paréntesis |
| **< { , - >** | Abre llave |
| **< } , - >** | Cierra llave |
| **< : , - >** | Dos puntos |
| **< . , - >** | Punto |
| **< , , - >** | Coma |
| **< sl , - >** | Salto de línea |
| **< $ , - >** | Fin de fichero |

Como se puede comprobar en la tabla, tanto los operadores como las palabras reservadas tienen códigos que generan distintos tokens englobados en la misma familia. Los códigos para los mismos son los siguientes:

* OpAritmeticos:

1. Mas ‘+’ 🡪 código = 1
2. Menos ‘-‘ 🡪 código = 2

* OpAsignación:

1. Igual ‘=’ 🡪 código = 1
2. MasIgual ‘+=’ 🡪 código = 2
3. MenosIgual ‘-=’ 🡪 código = 3

* OpLógico:

1. Negación ‘!’ 🡪 código = 1

* OpRelacional:

1. Menor o igual ‘<=’ 🡪 código = 1
2. Menor ‘<’ 🡪 código =2

Es decir, si generamos el token <OpAsignación, 2> en realidad lo que estamos consiguiendo es <OpAsignacion, +=>.

Por otra parte, el conjunto de palabras reservadas que procesará nuestro analizador será el siguiente:

***switch , case , break , return , if , function , prompt , document , write , var, default***

1. **Gramática:**

Una vez visto los tokens podemos proceder a generar la gramática correspondiente al analizador léxico. Esta gramática es la siguiente:

**Terminales = { l d " / + - ( ) { } : , . < sl $ \_ \* = ! lambda }**

**NoTerminales = { S PAL NUM CAD COMENT COMENT2 COMENT3 SUM RES MEN }**

**Axioma = S**

**Producciones = {**

**S 🡪 delS | lPAL | dNUM | "CAD | /COMENT | +SUM | -RES | ( | ) | { | } | : | , | . | <MEN | sl | $ | = | !**

**PAL 🡪 lPAL | dPAL | \_PAL | lambda**

**NUM 🡪 dNUM | lambda**

**CAD 🡪 ccCAD | "**

**COMENT 🡪 \*COMENT2**

**COMENT2 🡪 \*COMENT3 | ccCOMENT2**

**COMENT3 🡪 /S | ccCOMENT2**

**SUM 🡪 = | lambda**

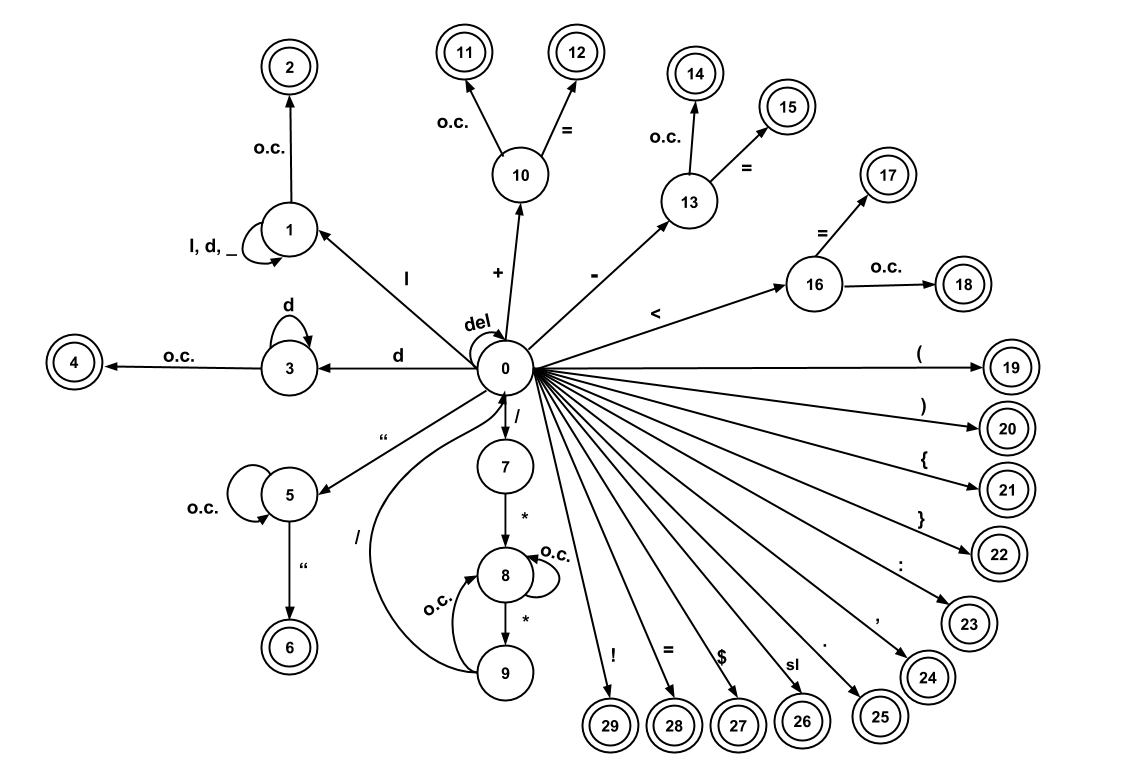
**RES 🡪 = | lambda**

**MEN 🡪 = | lambda**

**}**

1. **Autómata:**

EL autómata generado con la gramática anterior es el siguiente:



1. **Acciones Semánticas:**

Una vez creado el autómata, veremos paso a paso que ocurre en cada una de las transiciones, como y cuando se generan los tokens, y las comprobaciones previas que se realizan para poder generarlos. Las acciones semánticas son las siguientes:

**00:01-** lexema = carácter

**01:01-** lexema += carácter

**01:02-** si lexema = PReservada

🡪 Generar token(PReservada, lexema)

sino buscar lexema en TS, si no está 🡪 Añadir a TS y guardar posición

🡪 Generar token (id, posTS)

**00:03-** lexema = carácter

**03:03-** lexema += carácter

**03:04-** calcular valor lexema

🡪 Generar token (entero, valor)

**05:05-** lexema += carácter

**05:06-** 🡪 Generar token (cadena, lexema)

**00:10-** lexema = +

**10:11-** 🡪 Generar token (opArit, +)

**10:12-** lexema += =

🡪 Generar token (opAsig, +=)

**00:13-** lexema = -

**13:14-** 🡪 Generar token (opArit, -)

**13:15-** lexema += =

🡪 Generar token (opAsig, -=)

**00:16-** lexema = <

**16:17-** 🡪 Generar token (opLog, <)

**16:18-** lexema += =

🡪 Generar token (opAsig, <=)

**00:19-** 🡪 Generar token (abrePar)

**00:20-** 🡪 Generar token (cierraPar)

**00:21-** 🡪 Generar token (abreLLav)

**00:22-** 🡪 Generar token (cierraLLav)

**00:23-** 🡪 Generar token (dosPuntos)

**00:24-** 🡪 Generar token (coma)

**00:25-** 🡪 Generar token (punto)

**00:26-** 🡪 Generar token (finLinea)

**00:27-** 🡪 Generar token (finFichero)

**00:28-** 🡪 Generar token (opAsig, =)

**00:29-** 🡪 Generar token (opLog, !)

1. **Tablas de símbolos:**

Las tablas de símbolos se van almacenando en una pila. Esta pila inicialmente tendrá tan solo la tabla de símbolos global y el resto de tablas, es decir, cuando aparezca una declaración de función, se irán almacenando en lo alto de la pila, siendo la cima de la pila la tabla de símbolos actual en todo momento.

Las tablas de símbolos tienen además un controlador. Este controlador se encarga de crear y destruir tablas de símbolos, o de verificar si un lexema corresponde a un identificador o a una palabra reservada.

Cada vez que se define un identificador se comprueba si está o no en las tablas de símbolos. En caso de no estarlo se insertan en ella. Cuando se define una función, como ya se ha mencionado anteriormente, se crea una nueva tabla que se coloca en la cima de la pila y los siguientes identificadores se insertarán aquí hasta que salgamos de la función y se destruya la tabla, volviendo a la tabla anterior en la pila.

Las tablas de símbolos contienen información de cada elemento en ella. En caso de que el elemento insertado no pueda o no tenga alguno de estos campos se rellenará con un guión. La información que contienen es la siguiente:

* POS: Contiene la posición ocupada en la tabla de símbolos
* LEXEMA: Contiene el nombre del identificador, palabra reservada o de la función correspondiente.
* TIPO: Indica el tipo de datos.
* DESP: Desplazamiento en la pila.
* NUMPARAM: Número de parámetros que tiene (En caso de ser función).
* TIPODEV: Indica el tipo del valor que se devuelve (En caso de ser función).

Al finalizar la ejecución del procesador, si todo ha ido bien, las tablas de símbolos (tanto la global como las tablas de funciones) se imprimirán en un fichero de texto llamado *“tabla\_simbolos.txt”*.

1. **Analizador Sintáctico:**

El analizador sintáctico realizará un análisis ascendente LR. Este análisis genera una tabla GoTo y una tabla acción. Para conseguir estas tablas hemos utilizado la herramienta Anagra, la cual pasándole la gramática correspondiente al analizador sintáctico (en un formato específico) te genera automáticamente dos archivos, uno llamado *‘GoTo.dat’* con la tabla GoTo, y otro archivo llamado *‘decision.dat’* con la tabla de acción. Veamos pues la gramática del analizador sintáctico antes de explicar la manera en que analiza los datos.

* **Gramática:**

**NoTerminales = { PROGRAMA0 PROGRAMAS PROGRAMA BLOQUE FUNCION SALTO SALTO2**

**EXPRESION CUERPO SENTENCIA CASE CASE2 ARGUMENTOS ARGUMENTOS2**

**LLAMADAFUN LLAMADAFUN2 ARITMETICA SIMPLE RETURNVALUE }**

**Axioma = PROGRAMA0**

**Terminales = { var id if abrePar cierraPar switch abreLlave cierraLlave case break default**

**dosPuntos function igual masIgual menosIgual document punto write prompt**

**return sl coma menor menorIgual mas menos negacion num cad }**

**Producciones = {**

**PROGRAMA0 🡪 PROGRAMAS**

**PROGRAMAS 🡪 SALTO PROGRAMA**

**PROGRAMA 🡪 BLOQUE**

**PROGRAMA 🡪 FUNCION**

**PROGRAMA 🡪 BLOQUE SALTO2 PROGRAMA**

**PROGRAMA 🡪 FUNCION SALTO2 PROGRAMA**

**PROGRAMA 🡪 lambda**

**BLOQUE 🡪 var id**

**BLOQUE 🡪 SENTENCIA**

**BLOQUE 🡪 if abrePar EXPRESION cierraPar SENTENCIA**

**BLOQUE 🡪 if abrePar EXPRESION cierraPar SALTO abreLlave SALTO2 CUERPO cierraLlave**

**BLOQUE 🡪 switch abrePar EXPRESION cierraPar SALTO abreLlave SALTO2 CASE cierraLlave**

**CASE 🡪 default dosPuntos SALTO CUERPO**

**CASE 🡪 case EXPRESION dosPuntos SALTO CUERPO CASE2**

**CASE2 🡪 CASE**

**CASE2 🡪 break SALTO2 CASE**

**FUNCION 🡪 function id abrePar ARGUMENTOS cierraPar SALTO abreLlave SALTO2 CUERPO cierraLlave**

**SENTENCIA -> id igual EXPRESION**

**SENTENCIA -> id masIgual EXPRESION**

**SENTENCIA -> id menosIgual EXPRESION**

**SENTENCIA -> id abrePar LLAMADAFUN cierraPar**

**SENTENCIA -> document punto write abrePar EXPRESION cierraPar**

**SENTENCIA -> prompt abrePar id cierraPar**

**SENTENCIA -> return RETURNVALUE**

**CUERPO -> BLOQUE SALTO2 CUERPO**

**CUERPO -> lambda**

**SALTO -> sl SALTO**

**SALTO -> lambda**

**SALTO2 -> sl SALTO**

**RETURNVALUE -> EXPRESION**

**RETURNVALUE -> lambda**

**LLAMADAFUN -> EXPRESION LLAMADAFUN2**

**LLAMADAFUN -> lambda**

**LLAMADAFUN2 -> coma EXPRESION LLAMADAFUN2**

**LLAMADAFUN2 -> lambda**

**ARGUMENTOS -> id ARGUMENTOS2**

**ARGUMENTOS -> lambda**

**ARGUMENTOS2 -> coma id ARGUMENTOS2**

**ARGUMENTOS2 -> lambda**

**EXPRESION -> EXPRESION menor ARITMETICA**

**EXPRESION -> EXPRESION menorIgual ARITMETICA**

**EXPRESION -> ARITMETICA**

**ARITMETICA -> ARITMETICA mas SIMPLE**

**ARITMETICA -> ARITMETICA menos SIMPLE**

**ARITMETICA -> SIMPLE**

**SIMPLE -> negacion SIMPLE**

**SIMPLE -> abrePar EXPRESION cierraPar**

**SIMPLE -> id**

**SIMPLE -> num**

**SIMPLE -> cad**

**SIMPLE -> id abrePar LLAMADAFUN cierraPar**

**}**

Una vez sabemos cómo se consiguen las tablas GoTo y de acción y tenemos la gramática podemos proceder a explicar cómo trabaja nuestro analizador sintáctico. El analizador sintáctico carga las tablas desde los archivos generados previamente, pide un token al analizador léxico y ejecuta la opción que le señalen las tablas, ya sea reducir por una regla, desplazar o aceptar. En caso de llegar a este último se imprimirá un archivo con el parse de la ejecución y en caso contrario nuestro programa te avisará mediante un error de que ha habido un error sintáctico.

1. **Analizador Semántico**

Para realizar el analizador semántico hemos diseñado en primera instancia el esquema de traducción que se muestra a continuación:

PROGRAMA0 -> PROGRAMAS { if (PROGRAMAS.tipo = "ok") then SIN ERRORES; else ERROR }

PROGRAMAS -> SALTO PROGRAMA { PROGRAMAS.tipo = PROGRAMA.tipo }

PROGRAMA -> BLOQUE { PROGRAMA.tipo = BLOQUE.tipo }

PROGRAMA -> FUNCION { PROGRAMA.tipo = FUNCION.tipo }

PROGRAMA -> BLOQUE SALTO2 PROGRAMA { if(BLOQUE.tipo = "ok") then PROGRAMA.tipo = PROGRAMA1.tipo; else PROGRAMA.tipo = "error" }

PROGRAMA -> FUNCION SALTO2 PROGRAMA { if(FUNCION.tipo = "ok") then PROGRAMA.tipo = PROGRAMA1.tipo; else PROGRAMA.tipo = "error" }

PROGRAMA -> lambda { PROGRAMA.tipo = "ok" }

BLOQUE -> var id { BLOQUE.tipo = "ok" }

BLOQUE -> SENTENCIA { BLOQUE.tipo = SENTENCIA.tipo }

BLOQUE -> if abrePar EXPRESION cierraPar SENTENCIA { if(EXPRESION.tipo = "entero/logico") then BLOQUE.tipo = SENTENCIA.tipo; else BLOQUE.tipo = "error" }

BLOQUE -> if abrePar EXPRESION cierraPar SALTO abreLlave SALTO2 CUERPO cierraLlave { if(EXPRESION.tipo = "entero/logico") then BLOQUE.tipo = CUERPO.tipo; else BLOQUE.tipo = "error" }

BLOQUE -> switch abrePar EXPRESION cierraPar SALTO abreLlave SALTO2 CASE cierraLlave { if(EXPRESION.tipo = "entero/logico") then BLOQUE.tipo = CASE.tipo; else BLOQUE.tipo = "error" }

CASE -> default dosPuntos SALTO CUERPO { CASE.tipo = CUERPO.tipo }

CASE -> case EXPRESION dosPuntos SALTO CUERPO CASE2 { if(EXPRESION.tipo = "entero/logico" && CUERPO.tipo = "ok" && CASE2.tipo="ok") then CASE.tipo = "ok"; else CASE.tipo = "error" }

CASE2 -> CASE { CASE2.tipo = CASE.tipo }

CASE2 -> break SALTO2 CASE { CASE2.tipo = CASE.tipo }

FUNCION -> function id abrePar ARGUMENTOS cierraPar SALTO abreLlave SALTO2 CUERPO cierraLlave //{ if(ARGUMENTOS.tipo = "ok") then FUNCION.tipo = CUERPO.tipo; else FUNCION.tipo = "error"; Eliminar TS Local}

SENTENCIA -> id igual EXPRESION {if(id.tipo!=funcion)then id.tipo = EXPRESION.tipo, SENTENCIA.tipo = "ok"; else SENTENCIA.tipo="error" }

SENTENCIA -> id masIgual EXPRESION { if (id.tipo = "entero/logico" && EXPRESION.tipo = "entero/logico") then SENTENCIA.tipo = "ok"; else SENTENCIA.tipo = "error" }

SENTENCIA -> id menosIgual EXPRESION { if (id.tipo = "entero/logico" && EXPRESION.tipo = "entero/logico") then SENTENCIA.tipo = "ok"; else SENTENCIA.tipo = "error" }

SENTENCIA -> id abrePar LLAMADAFUN cierraPar { if(id.tipo = "funcion")then SENTENCIA.tipo = LLAMADAFUN.tipo; else SENTENCIA.tipo = "error" }

SENTENCIA -> document punto write abrePar EXPRESION cierraPar { if (EXPRESION.tipo != "error") then SENTENCIA.tipo = "ok"; else SENTENCIA.tipo="error" }

SENTENCIA -> prompt abrePar id cierraPar { if (id.tipo!=funcion)then SENTENCIA.tipo = "ok"; else SENTENCIA.tipo = "error" }

SENTENCIA -> return RETURNVALUE { if(RETURNVALUE.tipo != "ok", "error", "vacio") then TSG.funcion.val\_dev = RETURNVALUE.tipo, SENTENCIA.tipo = "ok"; else SENTENCIA.tipo = "error"}

CUERPO -> BLOQUE SALTO2 CUERPO { if (CUERPO1.tipo = "ok" && BLOQUE.tipo = "ok") then CUERPO.tipo = "ok"; else CUERPO.tipo = "error"}

CUERPO -> lambda { CUERPO.tipo = "ok" }

SALTO -> sl SALTO

SALTO -> lambda

SALTO2 -> sl SALTO

RETURNVALUE -> EXPRESION { RETURNVALUE.tipo = EXPRESION.tipo }

RETURNVALUE -> lambda { RETURNVALUE.tipo = "vacio" }

LLAMADAFUN -> EXPRESION LLAMADAFUN2 { if(EXPRESION.tipo = "ok" && LLAMADAFUN2.tipo="ok") then LLAMADAFUN.tipo = "ok"; else LLAMADAFUN.tipo = "error"}

LLAMADAFUN -> lambda { LLAMADAFUN.tipo = "ok" }

LLAMADAFUN2 -> coma EXPRESION LLAMADAFUN2 { if(EXPRESION.tipo != "error","vacio")then LLAMADAFUN2.tipo = LLAMADAFUN21.tipo; else LLAMADAFUN2.tipo = "error"}

LLAMADAFUN2 -> lambda { LLAMADAFUN2.tipo = "ok" }

ARGUMENTOS -> id ARGUMENTOS2 { ARGUMENTOS.tipo = ARGUMENTOS2.tipo }

ARGUMENTOS -> lambda { ARGUMENTOS.tipo = "ok" }

ARGUMENTOS2 -> coma id ARGUMENTOS21 { ARGUMENTOS2.tipo = ARGUMENTOS21.tipo }

ARGUMENTOS2 -> lambda { ARGUMENTOS2.tipo = "ok" }

EXPRESION -> EXPRESION menor ARITMETICA { if(EXPRESION1.tipo = entero/logico && ARITMETICA.tipo = entero/logico) then EXPRESION.tipo = entero/logico else EXPRESION.tipo = "error" }

EXPRESION -> EXPRESION menorIgual ARITMETICA { if(EXPRESION1.tipo = entero/logico && ARITMETICA.tipo = entero/logico) then EXPRESION.tipo = entero/logico else EXPRESION.tipo = "error" }

EXPRESION -> ARITMETICA { EXPRESION.tipo = ARITMETICA.tipo }

ARITMETICA -> ARITMETICA mas SIMPLE { if(ARITMETICA1.tipo = entero/logico && SIMPLE.tipo = entero/logico) then ARITMETICA.tipo = entero/logico else ARITMETICA.tipo = "error" }

ARITMETICA -> ARITMETICA menos SIMPLE { if(ARITMETICA1.tipo = entero/logico && SIMPLE.tipo = entero/logico) then ARITMETICA.tipo = entero/logico else ARITMETICA.tipo = "error" }

ARITMETICA -> SIMPLE { ARITMETICA.tipo = SIMPLE.tipo }

SIMPLE -> negacion SIMPLE { if(SIMPLE1.tipo = entero/logico) then SIMPLE.tipo = entero/logico; else SIMPLE.tipo = "error" }

SIMPLE -> abrePar EXPRESION cierraPar { SIMPLE.tipo = EXPRESION.tipo }

SIMPLE -> id { SIMPLE.tipo = buscaTipoTS(id.ent) }

SIMPLE -> num { SIMPLE.tipo = entero/logico }

SIMPLE -> cad { SIMPLE.tipo = cadena }

SIMPLE -> id abrePar LLAMADAFUN cierraPar { if(LLAMADAFUN.tipo = "ok" && id.tipo = "funcion") then SIMPLE.tipo = id.tipoDevuelto; else SIMPLE.tipo = "error"}

Una vez hecho esto, hemos implementado en código todas las acciones relacionándolas con la regla correspondiente, de este modo al reducir por cualquier regla en el analizador sintáctico buscamos las acciones asociadas a dicha regla y las ejecutamos.

La comprobación de tipos se ha podido realizar gracias a que hemos reproducido la pila de estados correspondiente al analizador LR. En dicha pila almacenamos los estados por los que pasamos y los tokens que se van analizando a medida que avanzamos. Estos elementos los hemos representado como objetos con atributos que nos permiten conocer su tipo y otros aspectos necesarios para realizar las comprobaciones pertinentes logrando así conocer si el programa es correcto o no desde el punto de vista semántico.

1. **Tratamiento de errores:**

Los errores se pueden localizar en cualquiera de las tres etapas del análisis ya sea en el léxico, si se encuentra un elemento que no se corresponde con ninguno de los tokens especificados, en el sintáctico, si la estructura del programa analizado no se corresponde con la definida por la gramática, o en el semántico, si hay elementos del lenguaje que no se están utilizando correctamente. Por esto hemos desarrollado un controlador de errores accesible desde cualquier etapa del análisis y que almacena los errores que se van encontrando para imprimirlos, al finalizar la ejecución, a un fichero errores.txt.

Se ha intentado en la medida de lo posible que al encontrar un error no se detenga la ejecución del análisis para así poder dar la mayor información posible acerca del programa. Con este fin, el analizador semántico al encontrar un error lo envía al controlador y continúa con su ejecución. En el caso de los analizadores léxico y sintáctico esto no es posible ya que un error impide que pueda continuar la ejecución del autómata correspondiente.

1. **Pruebas**

Dsgasga.

