JAVIER TOMÁS FERNÁNDEZ MARTÍN Y GABRIEL CHILLERÓN PEINADO

MEMORIA

DE PROCESADORES DE LENGUAJE

Contenido

[MEMORIA 1 2](#_Toc124324390)

[ESPECIFICACIÓN COMPLETA DE TODAS LAS CATEGORÍAS LÉXICAS DEL LENGUAJE 2](#_Toc124324391)

[MDD 3](#_Toc124324392)

[EJEMPLO DE AUTOMATA PR 3](#_Toc124324393)

[EJEMPLO DE SIMBOLO 4](#_Toc124324394)

[DESCRIPICIÓN DEL CÓDIGO 4](#_Toc124324395)

[MEMORIA 2 6](#_Toc124324396)

[GRAMÁTICA LIMPIA LL1 6](#_Toc124324397)

[TABLA DE PRIMEROS 7](#_Toc124324398)

[TABLA DE SEGUNDOS 7](#_Toc124324399)

[DESCRIPCIÓN DEL CÓDIGO 7](#_Toc124324400)

[MEMORIA 3 8](#_Toc124324401)

[ESQUEMA DE TRADUCCIÓN DIRIGIDO POR LA SINTAXIS 8](#_Toc124324402)

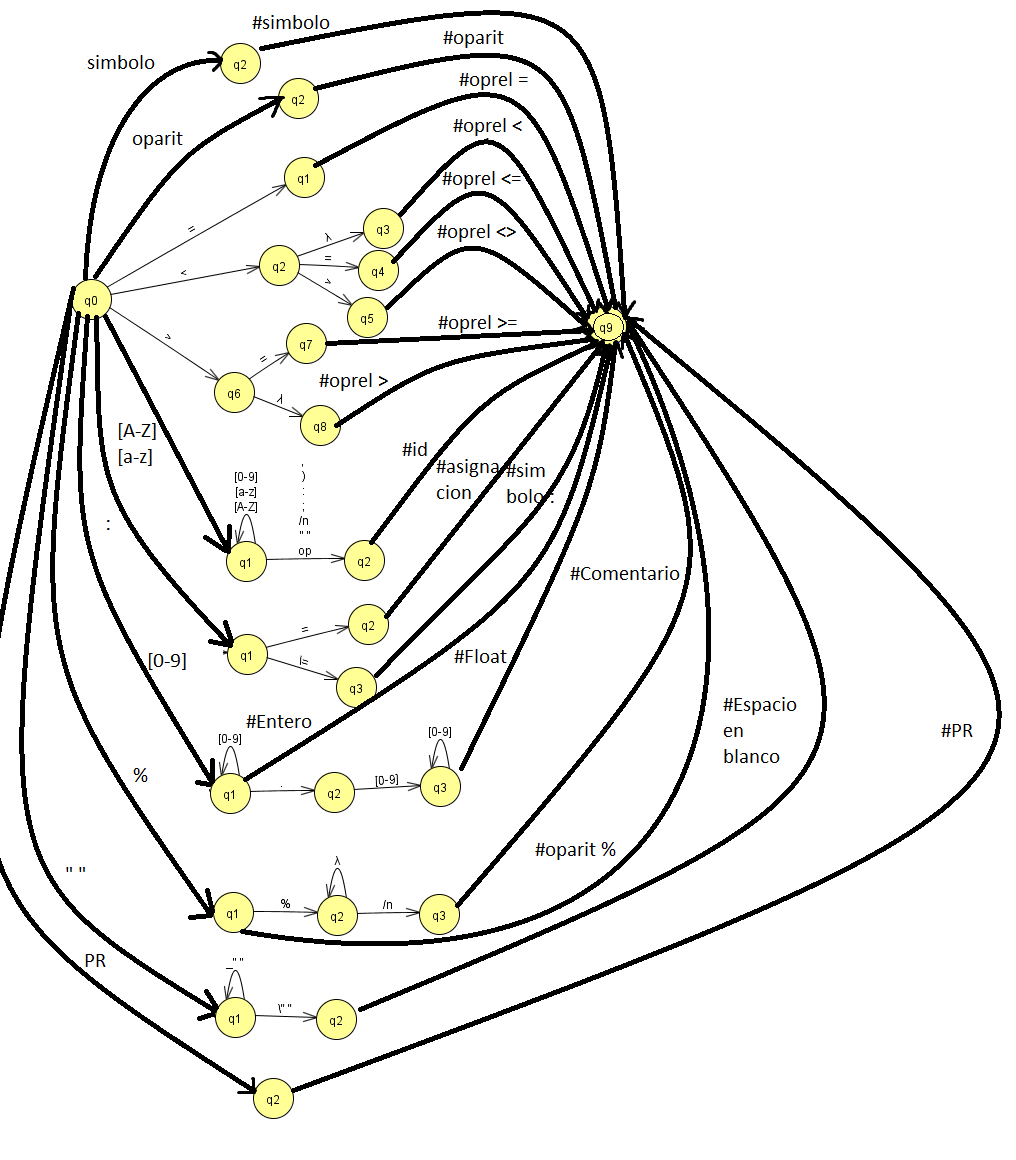
[DESCRIPCIÓN DEL CÓDIGO 12](#_Toc124324403)

# MEMORIA 1

## ESPECIFICACIÓN COMPLETA DE TODAS LAS CATEGORÍAS LÉXICAS DEL LENGUAJE

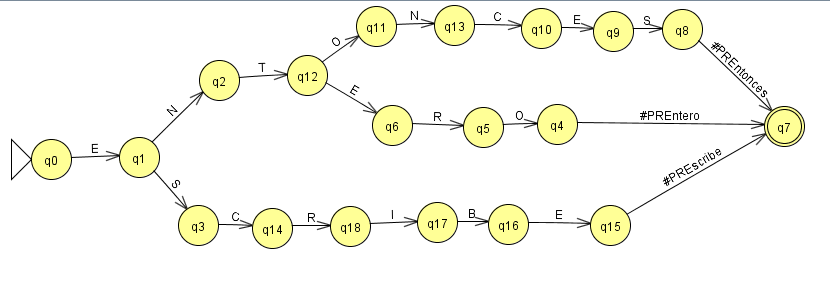
|  |  |  |
| --- | --- | --- |
| Categoría léxica | Lexemas | Patrón |
| PR | PROGRAMA, VAR, ENTERO, REAL, BOOLEANO, INICIO, FIN, SI, ENTONCES, SINO, MIENTRAS, HACER, LEE, ESCRIBE, Y, O, NO, CIERTO y FALSO | Cualquiera de las palabras mencionadas anteriormente. Debe estar en mayúsculas. |
| OpAsigna | := | Carácter “:” seguido de “=” |
| OpAdd | + - | Carácter “+” o “-“ |
| OpMult | / \* | Carácter “\*” o “/” |
| Numero | 4 1023 45.2 | Secuencia de uno o más dígitos, con posibilidad de poner un punto y poner otra secuencia de uno o más dígitos |
| Identif | za3fS, ZER3sd | Cadenas de letras y números que empiecen por una letra. |
| OpRel | < <> > = >= <= | Caracteres “<”, “<>”, “>”, “=”, “>=” o “<=” |
| Símbolo | ; ( ) : , | Caracteres “;”, “(”, “)”, “:” o “,” |
| Comentario | %%asdSA0132 | Secuencia de dos “%” seguido de cualquier cosa |

## MDD

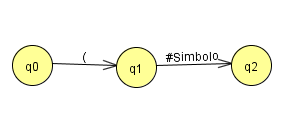


En el MDD los espacios se han representado como " ", los "distinto de espacio" como \" ", al igual que los distintos del símbolo igual “\=”. Por último, para simplificar un poco el problema, en las palabras reservadas (PR), sería necesario un autómata por palabra reservada que leyese letra por letra y comprobase si es una PR o no. Ese autómata se ha agregado en el estado al que se va con PR, donde se trataría y mandaría la salida al último estado. Se ha hecho de igual forma con los símbolos "(" , ")" , ";" y ",". Se han englobado todos los casos en uno único, ya que dichos autómatas tendrían solo un estado y sería un caos meterlo también en el MDD.

### EJEMPLO DE AUTOMATA PR



### EJEMPLO DE SIMBOLO



# DESCRIPICIÓN DEL CÓDIGO

El objetivo de la primera parte de la práctica 3 es separar todos los componentes que representan una categoría léxica de nuestro programa principal. Para ello, vamos a utilizar el fichero componentes.py, donde vienen representadas todas las clases., y la clase Analex, donde se encuentra la lógica del analizador léxico.

Para realizar la lectura del fichero vamos a utilizar una clase flujo.py, que va a tener como atributo la clase Analex. Nosotros utilizamos dos funciones de esta clase; ***siguiente (),*** que nos devuelve el próximo carácter de la cadena y ***devuelve (c)*** que introduce “c” en la cadena y retrocede el puntero.

Vamos a definir todas las funciones que contiene la clase:

* ***TrataNum (flujo, ch):*** Se encarga de leer la categoría léxica número. Además, identifica si el numero es de tipo float o de tipo entero. Para representar el tipo del número, se pasará como parámetro al constructor del componente <type 'float'> o <type 'int'> dependiendo de cada caso.
* **TrataIdent(flujo, ch):** Se encarga de leer los identificadores y las palabras reservadas. Hay que aclarar que para que la condición que utilizamos en el while no sea demasiado larga, utilizamos el código ASCII.
* ***TrataComent(flujo):*** Lee los comentarios del lenguaje. Como los comentarios son líneas que simplemente se deben ignorar, en esta función no es necesario devolver ningún componente.
* ***EliminaBlancos(flujo):*** Elimina los espacios en blanco de toda la línea en la que se encuentra el flujo.
* ***Analiza ():*** Es la función principal de la clase, se va a encargar, por decirlo de alguna manera, de seleccionar el camino que debe tomar la MDD***.*** Por ejemplo, lo primero que nosotros hacemos es comprobar si lo que llega al estado inicial es un número, lo que equivaldría a comprobar en la MDD si la línea que parte de q0 es [0-9]. Si es verdad que es un número, llamamos a TrataNum(), que sería el equivalente al autómata para tratar los números. Este proceso lo repetimos para cada una de las líneas que parten de q0 en la MDD. Por último la función main del programa se va a encargar de llamar recurrentemente a esta función hasta que devuelve el componente EOF, que representa el final del fichero.

# MEMORIA 2

## GRAMÁTICA LIMPIA LL1

<Programa> → PROGRAMA id ; <decl\_var> <instrucciones> .

<decl\_var> → VAR <lista\_id>: <tipo\_std> ; <decl\_v>|λ

<decl\_v>→ <lista\_id>: <tipo\_std>; <decl\_v>|λ

<lista\_id>→ id <resto\_listaid>

<resto\_listaid>→ , <lista id>|λ

<Tipo\_std>→ ENTERO | REAL | BOOLEANO

<instrucciones>→ INICIO <lista\_inst>FIN

<lista\_inst>→ <instruccion>; <lista\_inst>|λ

<instruccion>→ INICIO <lista\_inst>FIN |

<instruccion>→ <inst\_simple>|

<instruccion>→ <inst\_e/s>

<instruccion>→ SI <expresion> ENTONCES <instrucción> SINO <instruccion>

<instruccion>→ MIENTRAS <expresion> HACER <instruccion>

<Inst\_simple>→ id opasigna <expresion>

<inst\_e/s>→ LEE ( id ) | ESCRIBE ( <expr\_simple>)

<expresion>→ <expr\_simple> <expresion2>

<expresion2>→ oprel <expr\_simple> | λ

<expr\_simple> → <termino><resto\_exsimple>

<expr\_simple> → <signo> <termino> <resto\_exsimple>

< resto\_exsimple>→ opsuma <termino><resto\_exsimple>

< resto\_exsimple>→ O <termino><resto\_exsimple>|λ

<termino> → <factor><resto\_term>

<resto\_term>→ opmult <factor><resto\_term>

<resto\_term>→ Y <factor><resto\_term>|λ

<factor>→ <id> | num | ( <expresion> ) | NO <factor> | CIERTO | FALSO

<signo>→ +|−

### TABLA DE PRIMEROS

<Programa> → {PROGRAMA}

<decl\_var> → {VAR, λ}

<decl\_v>→ {Id,λ}

<lista\_id>→ {id}

<resto\_listaid>→ {\, , λ}

<Tipo\_std>→ {ENTERO, REAL, BOOLEANO}

<instrucciones>→ {INICIO}

<lista\_inst>→ {λ}

<instruccion>→ {INICIO, SI, MIENTRAS, Id, LEE, ESCRIBE}

<Inst\_simple>→ {id}

<inst\_e/s>→ {LEE, ESCRIBE}

<expresion>→ {Id, num, (, NO, CIERTO, FALSO, +, −}

<expresion2>→ {oprel , λ}

<expr\_simple> → {Id, num, (, NO, CIERTO, FALSO, +, −}

< resto\_exsimple>→ {opsuma, O, λ }

<termino> → {Id, num, (, NO, CIERTO, FALSO}

<resto\_term>→ {opmult ,Y,λ }

<factor>→ {Id, num, (, NO, CIERTO, FALSO}

<signo>→ {+, −}

### TABLA DE SIGUIENTES

<Programa> → {$}

<decl\_var> → {INICIO}

<decl\_v>→ {INICIO}

<lista\_id>→ { : }

<resto\_listaid>→ {: }

<Tipo\_std>→ { ; }

<instrucciones>→ {INICIO}

<lista\_inst>→ {FIN, INICIO }

<instruccion>→ {;, SINO }

<Inst\_simple>→ {}

<inst\_e/s>→ {}

<expresion>→ {ENTONCES, HACER}

<expresion2>→ { ENTONCES, HACER }

<expr\_simple> → { ), oprel }

< resto\_exsimple>→ {), oprel }

<termino> → { opsuma }

<resto\_term>→ {opmult, opsuma, Id, num, (, NO, CIERTO, FALSO}

<factor>→ {Id, num, (, NO, CIERTO, FALSO}

<signo>→ {Id, num, (, NO, CIERTO, FALSO}

### DESCRIPCIÓN DEL CÓDIGO

El objetivo del programa que creamos para la segunda parte de la práctica 3 va a ser el comprobar que los tokens generados por el analizador léxico llegan en el orden correcto, es decir, si la cadena del programa podría ser generada a partir de la gramática.

Para ello vamos a definir una función por cada producción que tenemos en nuestra gramática. Estas funciones van a devolver una tupla de valores. El segundo valor de la tupla va a corresponder a la tercera parte de la práctica, por lo que será explicado en esta sección de la memoria. El primer valor de la tupla corresponde con un booleano, que será False cuando se analice un token que no puede ser generado por ninguna de las producciones de la función.

Por ejemplo, la función signo puede producir dos terminales, “+” o “-”. Estos dos símbolos pertenecen a la categoría léxica “OpAdd”, por lo que nuestro analizador léxico devolverá un componente de tipo OpAdd. Lo que haremos en el analizador sintáctico es que, cuando se llame a esta función, comprobaremos si el token es del tipo de ese Componente. En caso negativo, mostraremos un mensaje de error indicando que se esperaba un + o un -, y devolvemos False. En caso de que el componente sea el correcto, lo que haremos será actualizar el valor del token con la operación token.Analiza(), para que lea el siguiente componente y devolver un valor True.

Así procederíamos con todas las producciones de nuestra gramática, teniendo en cuenta que si una producción contiene a un no terminal habría que llamar a la función que representa ese no terminal y retornar el booleano que produzca. También hay que tener en cuenta que, en las producciones anulables, si llega un componente que ellas no generan, puede ser que sea precisamente porque son anulables. En esos casos devolvemos también el valor True.

# MEMORIA 3

## ESQUEMA DE TRADUCCIÓN DIRIGIDO POR LA SINTAXIS

<Programa> → PROGRAMA id ; <decl\_var> <instrucciones> .

{lsen = [<decl\_var>.arbol, <instrucciones>.arbol]

aux = NodoCompuesta(lsen, self.token.linea)

<Programa>.arbol = NodoPrograma(id.valor, aux, self.token.linea) }

<decl\_var> → VAR <lista\_id>: <tipo\_std> ; <decl\_v>|λ

<decl\_v>→ <lista\_id>: <tipo\_std>; <decl\_v>|λ

<lista\_id>→ id <resto\_lista\_id>

{<lista\_id>.arbol =

<resto\_listaid>→ , <lista\_id>

{<resto\_listaid>.arbol = <lista\_id>.arbol}

<resto\_listaid>→ λ

{<resto\_listaid>.arbol = []}

<Tipo\_std>→ ENTERO | REAL | BOOLEANO

<instrucciones>→ INICIO <lista\_inst>FIN

{nodoLista = NodoCompuesta(<lista\_inst>.arbol, self.token.linea)

<instrucciones>.arbol = NodoInicio(nodoLista, self.token.linea) }

<lista\_inst>→ <instruccion>; <lista\_inst>

{<lista\_inst>1.arbol = [<instruccion>.arbol, <lista\_inst>2.arbol] }

<lista\_inst>→ λ

{<lista\_inst>.arbol = []}

<instruccion>→ INICIO <lista\_inst>FIN

{<instruccion>.arbol = NodoInicio(<lista\_inst>.arbol,self.token.linea}

<instruccion>→ <inst\_simple>

{<instruccion>.arbol = <inst\_simple>.arbol}

<instruccion>→ <inst\_e/s>

{<instruccion>.arbol = <inst\_e/s>.arbol}

<instruccion>→ SI <expresion> ENTONCES <instrucción> SINO <instruccion>

{<instruccion>1.arbol = NodoSi(<expresion>.arbol, <instruccion>2.arbol, <instruccion>3.arbol, self.token.linea}

<instruccion>→ MIENTRAS <expresion> HACER <instruccion>

{<instruccion>.arbol = NodoMientras(<expresion>.arbol, <instruccion>.arbol, self.token.linea}

<inst\_simple>→ id opasigna <expresion>

<inst\_simple>.arbol = NodoAsignacion(id.valor, <expresion>.arbol, self.token.linea)}

<inst\_e/s>→ LEE ( id )

{<inst\_e/s>.arbol = NodoLee(id.valor, self.token.linea)}

<inst\_e/s>→ ESCRIBE ( <expr\_simple>)

{<inst\_e/s>.arbol = NodoEscribe(<expr\_simple>.arbol, self.token.linea)}

<expresion>→ <expr\_simple> <expresion2>

{if <expresion2>.arbol == [] then {

<expresion>.arbol = NodoExpresion(<expr\_simple>.arbol ,self.token.linea)

} else {

aux =NodoRelacion(<expr\_simple>.arbol, <expresion2>.arbol, self.token.linea, <expresion2>.op)

<expresion>.arbol =NodoExpresion(aux,self.token.linea)

}}

<expresion2>→ oprel <expr\_simple>

{<expresion2>.arbol = <expr\_simple>.arbol

<expresion2>.op = oprel.valor}

<expresion2>→ λ

{<expresion2>.arbol = []}

<expr\_simple> → <termino><resto\_exsimple>

{if <resto\_ exsimple >.arbol == [] then {

<expr\_simple >.arbol = <termino>.arbol

} else {

<expr\_simple >.arbol = NodoAritmetico(<termino>.arbol, <resto\_exsimple >.arbol, self.token.linea, <resto\_exsimple>.op)

}}

<expr\_simple> → <signo> <termino> <resto\_exsimple>

{if <resto\_ exsimple >.arbol == [] then {

<expr\_simple >.arbol = <termino>.arbol

} else {

<expr\_simple >.arbol = NodoAritmetico (<signo>.signo, <termino>.arbol, <resto\_exsimple >.arbol, self.token.linea, <resto\_exsimple>.op)

}}

< resto\_exsimple>→ opsuma <termino><resto\_exsimple>

{if <resto\_ exsimple >2.arbol == [] then {

<resto\_ exsimple >1.arbol = <termino>.arbol

} else {

<resto\_ exsimple >1.arbol = NodoAritmetico (<termino>.arbol, <resto\_ exsimple >2.arbol, self.token.linea, <resto\_ exsimple >2.op)

}

<resto\_ exsimple >1.op = opsuma.valor}

< resto\_exsimple>→ O <termino><resto\_exsimple>

{if <resto\_ exsimple >2.arbol == [] then {

<resto\_ exsimple >1.arbol = <termino>.arbol

} else {

<resto\_ exsimple >1.arbol = NodoAritmetico (<termino>.arbol, <resto\_ exsimple >2.arbol, self.token.linea, <resto\_ exsimple >2.op)

}

<resto\_ exsimple >1.op = O.valor}

< resto\_exsimple>→ λ

{<resto\_exsimple>.arbol = []}

<termino> → <factor><resto\_term>

{if <resto\_term>.arbol == [] then {

<termino>.arbol = <factor>.arbol

} else {

<termino>.arbol = NodoAritmetico(<factor>.arbol, <resto\_term>.arbol, self.token.linea, <resto\_term>.op)

}}

<resto\_term>1.arbol = NodoAritmetico (<factor>.arbol, <resto\_term>2.arbol, self.token.linea, <resto\_term>2.op

<resto\_term>→ opmult <factor><resto\_term>

{if <resto\_term>2.arbol == [] then {

<resto\_term>1.arbol = <factor>.arbol

} else {

<resto\_term>1.arbol = NodoAritmetico (<factor>.arbol, <resto\_term>2.arbol, self.token.linea, <resto\_term>2.op)

}

<resto\_term>1.op = opmult.valor}

<resto\_term>→ Y <factor><resto\_term>

{if <resto\_term>2.arbol == [] then {

<resto\_term>1.arbol = <factor>.arbol

} else {

<resto\_term>1.arbol = NodoAritmetico (<factor>.arbol, <resto\_term>2.arbol, self.token.linea, <resto\_term>2.op

}

<resto\_term>1.op = Y.valor }

<resto\_term>→ λ

{<resto\_term>.arbol = []}

<factor>→ <id>

{<factor>.arbol = NodoId(id.valor, self.token.linea}

<factor>→ num

{if num.tipo == “<type ‘float’>” then <factor>.arbol = NodoReal (num.valor, self.token.linea)

else <factor>.arbol = NodoEntero (num.valor, self.token.linea) }

<factor>→ ( <expresion> )

{<factor>.arbol = <expresion>.arbol}

<factor>→ NO <factor>

{<factor>1.arbol = NodoNegación(<factor>2.arbol,self.token.linea)}

<factor>→ CIERTO

{<factor>.arbol = NodoBooleano(1,self.token.linea)}

<factor>→ FALSO

{<factor>.arbol = NodoBooleano(0,self.token.linea)}

<signo>→ +

{<signo>.valor = +}

<signo>→ −

{<signo>.valor = −}

### DESCRIPCIÓN DEL CÓDIGO

La última parte de la práctica 3 va a consistir en la construcción de los árboles de sintaxis abstracta, y en la comprobación de ciertas restricciones semánticas.

Para construir los AST, lo que haremos será devolver en las funciones, como segundo valor de la tupla, los atributos indicados en el esquema de traducción dirigido por la sintaxis. Construiremos los nodos, que vienen definidos en el fichero AST.py, en las funciones que corresponda. Por ejemplo, el nodo REAL y el nodo ENTERO se van a formar cuando llegue un componente de tipo Número a la función factor. Este nodo generado se propaga hacia arriba, ya que podría formar parte de otro nodo superior.

Para cumplir con las restricciones semánticas hemos decidido lanzar una excepción, con una pequeña descripción de lo que ha ocurrido, cuando no se cumpla alguna de ellas. También hemos necesitado decir como atributo del Analex un diccionario que sirva como tabla de símbolos. Para identificar los fallos hemos tratado cada una de la siguiente manera:

* **R1**: Cuando se define el identificador, tanto en decl\_var como en decl\_v, se comprueba si ya estaba previamente definido en la tabla de símbolos. En caso afirmativo, se lanza la excepción. En el caso de que estuviese, se añade.
* **R2**: Comprobamos que los ids generados en lista\_id no pertenecen al conjunto de palabras reservadas.
* **R3**: Esta restricción no tenemos claro que significa, lo que nosotros hemos hecho es tratar los enteros como reales. Al crear un entero, le añadimos “.0” para que sea un real.
* **R4**: Internamente nos guardamos en el valor de los nodos Booleanos 0, si llega FALSO y 1, si llega CIERTO. Esta acción interna la realizamos en la función compsem del nodo BOOLEANO.
* **R5**: Tal como lo tenemos planteado, no se podrá guardar 0 y 1 como booleano, aunque internamente sí que lo haga.
* **R6**: Se comprueban en las funciones compsem de los respectivos nodos. En NodoLee se comprueba que el valor es REAL o ENTERO, y en NodoEscribe se comprueba que exp es un nodo de tipo NodoExpresión.