

# ORGANIZACIÓN DE LENGUAJES Y COMPILADORES 1

[illegible]

# TRADUCTOR C# A PYTHON

**FRANCISCO LUIS SUAREZ LÓPEZ**

**201807190**



# **Desarrollo de la aplicación**

# ANALISIS LEXICO

```
//funcion para el analisis lexico
function analisis_lexico(entrada) {
    //Iniciamos las listas
    Lista de Tokens = [];
    Lista de Errores = [];
    let estado = 0;
    let columna = 0;
    let fila = 1;
    let lexema = "";
    let c = "";
    entrada = entrada + " ";
    let contador_caracter = 0;
    pasolibre = true;

    //Empezamos el analisis
    for (let i = 0; i < entrada.length; i++) {
        c = entrada[i];
        columna++;
        switch (estado) {
            case 0:
                //Revisara si puede ser una palabra reservada, un caracter o una variable
                if (isLetter(c)) {
                    estado = 1;
                    lexema += c;
                }
                //Revisara si puede ser un numero
                else if (isDigit(c)) {
                    estado = 2;
                    lexema += c;
                }
                //Revisara si puede ser una cadena
                else if (c === '"') {
                    estado = 5;
                    i--;
                    columna--;
                }
            }
        }
    }
}
```

El análisis léxico consta de recorrer la cadena de texto que se le envía como parámetro, en donde se lleva el control de la fila, la columna y el tipo de token que se reconoce a partir de los patrones reconocidos por el lenguaje establecido.

A partir de este análisis se genera una lista de tokens encontrados y una lista de errores, estos son tokens que el lenguaje no reconoce como parte de él, con lo cual no son admitidos.

Entre los diferentes patrones que se reconocen están los siguientes:

- Si se comienza con una letra, puede ser una palabra reservada del lenguaje o un identificador, los cuales se representan de la siguiente forma

Letra . (Letra | Dígito | `_`) \*

- Si se identifica un dígito, puede que sea un número entero o un número decimal, los cuales se representan de la siguiente forma

(Dígito)+ . (.)? . (Dígito)+

- Si se identifica el uso de comillas dobles, todo lo que venga después de eso se tomara como una cadena de texto y dejara de tomarlo hasta encontrar otras comillas dobles.

```
595     case 5:
596         //Comprueba que es una cadena
597         lexema += c;
598         estado = 6;
599         break;
600     case 6:
601         //Compraba todos los datos que contendrá la cadena, hasta encontrar otro (") para cerrar la cadena
602         if (c === '"') {
603             columna = 0;
604             fila++;
605             estado = 6;
606         } else if (c !== '"') {
607             lexema += c;
608             estado = 6;
609         } else {
610             estado = 7;
611             i--;
612             columna--;
613         }
614     }
615     break;
616     case 7:
617         //aquí cierra la cadena al encontrar (")
618         lexema += c;
619         addToken("Cadena", lexema, fila, columna);
620         estado = 0;
621         lexema = "";
622         break;
```

- Si se identifican comillas simples, puede que se trate de un carácter o de una cadena de html y esto deja de representarse como tal hasta encontrar la otra comilla simple.

```

849 case 18:
850 //Comproba que es un caracter lo que viene viendo si es comilla simple
851 contador_Caracter = 0;
852 if (c == "'") {
853     lexema += c;
854     estado = 19;
855 }
856 break;
857 case 19:
858 //Comprobar el caracter que viene para asignarle y cuando encuentre el otro (') cierra
859 if (c != "'") {
860     contador_Caracter++;
861     lexema += c;
862     estado = 19;
863 } else if (contador_Caracter < 2) {
864     estado = 20;
865     i--;
866     columna--;
867 } else if (contador_Caracter >= 2) {
868     estado = 21;
869     i--;
870     columna--;
871 }
872 break;
873 case 20:
874 //Aquí cierra la asignacion del carater
875 if (c == "'") {
876     lexema += c;
877     addToken("Caracter", lexema, fila, columna);
878     estado = 0;
879     lexema = "";
880 }
881 break;
882 case 21:
883 //Comprobar todos los datos que contendra la cadena de HTML, hasta encontrar otro (') para cerrar la cadena
884 if (c == "<") {
885     columna = 0;
886     fila++;
887     estado = 21;
888 } else if (c != "'") {
889     lexema += c;
890     estado = 21;
891 } else {
892     estado = 22;
893     i--;
894     columna--;
895 }
896 break;
897 case 22:
898 //Aquí cierra la cadena al encontrar (')
899 lexema += c;
900 addToken("Cadena html", lexema, fila, columna);
901 estado = 0;
902 lexema = "";
903 break;

```

- En dado caso puede ser uno de los muchos símbolos admitidos por el lenguaje.

```

312 //Lista de tokens ya establecidos que son todos los símbolos admitidos
313 else if (c == "(") {
314     lexema += c;
315     addToken("llave izquierda", lexema, fila, columna);
316     lexema = "";
317 } else if (c == ")") {
318     lexema += c;
319     addToken("llave derecha", lexema, fila, columna);
320     lexema = "";
321 } else if (c == "[") {
322     lexema += c;
323     addToken("corchete izquierdo", lexema, fila, columna);
324     lexema = "";
325 } else if (c == "]") {
326     lexema += c;
327     addToken("corchete derecho", lexema, fila, columna);
328     lexema = "";
329 } else if (c == "{") {
330     lexema += c;
331     addToken("Parentesis izquierdo", lexema, fila, columna);
332     lexema = "";
333 } else if (c == "}") {
334     lexema += c;
335     addToken("Parentesis derecho", lexema, fila, columna);
336     lexema = "";
337 } else if (c == ",") {
338     lexema += c;
339     addToken("Coma", lexema, fila, columna);
340     lexema = "";
341 } else if (c == ";") {
342     lexema += c;
343     addToken("Punto y coma", lexema, fila, columna);
344     lexema = "";
345 } else if (c == "=") {
346     estado = 9;
347     lexema += c;
348 } else if (c == ".") {

```

- Para culminar, como se dijo anteriormente, si no es ningún de estos patrones, quiere decir que el lexema a reconoces no es admitido en el lenguaje, lo cual se ingresa como error léxico.

```

904         default:
905             lexema += c;
906             if (
907                 c === "\n" ||
908                 c === "\\\" ||
909                 c === "\t" ||
910                 c === " " ||
911                 c.charCodeAt(0) === 13 ||
912                 c.charCodeAt(0) === 9
913             ) {
914                 estado = 0;
915                 lexema = "";
916             } else {
917                 addError("Desconocido", lexema, fila, columna);
918                 estado = 0;
919                 lexema = "";
920                 pasoLibre = false;
921             }
922             break;

```

## ANALISIS SINTACTICO

```

978 //funcion del parser
979 function parser() {
980     //Vamos a añadir un ultimo token para saber donde termina
981     addToken("ultimo", "ultimo", "0", "0");
982     //Reiniciamos las listas
983     ListaVariables = [];
984     ListaErroresSintacticos = [];
985     //Reiniciamos todos los valores
986     indice = 0;
987     tokenActual = Lista_de_Tokens[indice];
988     errorSintactico = false;
989     Error_Sintactico_Permiso = true;
990     isVoid = false;
991     //codigo traducido python
992     Contador_Tabs_Python = 0;
993     codigo_Python = "";
994     //llamado al no terminal inicial
995     inicio();
996 }

```

El análisis sintáctico consta de recorrer la lista de tokens identificados en el análisis léxico, a partir de un parser, en este caso se utilizó una gramática de LL1 para poder realizar el análisis sintáctico.

El parser consta de una función emparejar (match), que lo que hace es comparar el tipo del token que fue detectado y el tipo de token esperado, si este da true, sigue el análisis, de lo contrario se encontró un error sintáctico ya que se esperaba cierto tipo de token y se identificó otro.

```

2058 function emparejar(tip) {
2059     if (tokenActual.Tipo !== "ultimo") {
2060         if (errorSintactico) {
2061             Error_Sintactico_Permiso = false;
2062             if (tokenActual.Tipo !== "ultimo") {
2063                 console.log(tokenActual);
2064                 indice++;
2065                 tokenActual = Lista_de_Tokens[indice];
2066                 if (
2067                     tokenActual.Tipo === "Punto y coma" ||
2068                     tokenActual.Tipo === "Llave derecha"
2069                 ) {
2070                     errorSintactico = false;
2071                 }
2072             } else {
2073                 if (tokenActual.Tipo !== "ultimo") {
2074                     if (tokenActual.Tipo === tip) {
2075                         indice++;
2076                         tokenActual = Lista_de_Tokens[indice];
2077                     } else {
2078                         addErrorSintactico(
2079                             tip,
2080                             tokenActual.Tipo,
2081                             tokenActual.Fila,
2082                             tokenActual.Columna
2083                         );
2084                     }
2085                     errorSintactico = true;
2086                 }
2087             }
2088         }
2089     }

```

Se implemento el método pánico para la recuperación de errores sintácticos, este funciona de forma que cuando se encuentra un error sintáctico, dejar de realizar match con los demás tipos que deberían venir hasta encontrar un tipo de token que sirva como bandera, en este caso se utilizó el tipo (Punto y coma ;) y el tipo (Llave derecha } ) para la recuperación.

A continuación, se adjunta la gramática LL1 utilizada para la práctica.

## GRAMATICA LL1 UTILIZADA

```

<Inicio>:= <Comentarios > Class ID { <Declaracion_Contenidos> } <Comentarios >
<Declaracion_Contenidos>:= <Lista_Declaracion><Declaracion_Contenidos>
                             | <Metodos ><Declaracion_Contenidos>
                             | <Comentarios ><Declaracion_Contenidos>
                             | Epsilon
<Lista_Declaracion>:= <Declaracion_Var><Lista_Declaracion>
                     | Epsilon
<Declaracion_Var>:= <Tipo> ID <Lista_IDP><Opcion_Asignacion>;
                   | ID <Lista_IDP> <Opcion_Asignacion> ;
                   | <Lista_ID>
<Tipo>:= int
        | double
        | char
        | bool
        | string
<Lista_ID>:= ID ( <Parametros> ) { <Sentencias> <Return> }
           | ID ( <IDP> ) ;
           | ID -- ;
           | ID ++ ;
<Lista_IDP>:= , ID <Lista_IDP>
            | Epsilon
<Parametros>:= <Tipo> ID
              | <Tipo> ID, <Parametros>
              | Epsilon
<Opcion_Asignacion>:= = <Expresiones>
                    | Epsilon
<Expresiones>:= <T><EP>
<EP>:= + <T><EP>
      | - <T><EP>
      | Epsilon
<T>:= <K><TP>
<TP>:= * <K><TP>
      | / <K><TP>
      | Epsilon
<K>:= <F><KP>
<KP>:= < <F><KP>

```

```

    | > <F><KP>
    | == <F><KP>
    | != <F><KP>
    | >= <F><KP>
    | <= <F><KP>
    | && <F><KP>
    | || <F><KP>
    | Epsilon
<F>:= Numero
    | Cadena
    | Caracter
    | Cadena HTML
    | <NotF>
<NotF>:= <!> ( <Expresiones> )
    | <!> true
    | <!> false
    | <!> ID <IDP>
<!>:= ! <!>
    | Epsilon
<IDP>:= ( )
    | ( <ExpF> )
    | Epsilon
<ExpF>:= <Expresiones>
    | <Expresiones> , <ExpF>
<Metodos> := void <Main>
    | void <Metodo_Void>
<Main>:= main ( ) { <Sentencias> <Return> }
<Metodo_Void>:= ID ( <Parametros> ) { <Sentencias> <Return> }
<Sentencias>:= <Lista_Declaracion><Sentencias>
    | <Comentarios><Sentencias>
    | <Imprimir><Sentencias>
    | <Condicionales><Sentencias>
    | <Ciclos><Sentencias>
    | Epsilon
<Imprimir>:= Console . Write ( <Expresiones> ) ;
<Condicionales>:= <If><Else>
    | <Switch>
<If>:= if ( <Expresiones> ) { <Sentencias> <Final_Sentencia> }
<Else>:= else <ElseP>
    | Epsilon
<ElseP>:= <If><Else>
    | { <Sentencias> <Final_Sentencia> }
<Switch>:= switch ( ID ) { <Cases> <Default> }
<Cases>:= case <Expresiones> : <CasesP>
    | Epsilon
<CasesP>:= <Cases>
    | <Sentencias> <Final_Sentencias> <Cases>

```

```

<Default>:= default : <Sentencias> <Final_Sentencias>
                | Epsilon
<Final_Sentencias>:= <Return>
                    | <Break>
                    | <Continue>
                    | Epsilon
<Return>:= return ;
                | return <Expresiones> ;
<Break>:= break ;
<Continue>:= continue ;
<Ciclos>:= <For>
            | <While>
            | <Do_While>
<For>:= for ( <Declaracion_Var> <Expresion_For> <Alter_For> { <Sentencias>
<Final_Sentencia> }
<Expresion_For>:= ID <Simbolo_For> <Expresiones>
                | <Expresiones> <Simbolo_For> ID
<Simbolo_For>:= <
                | >
                | ==
                | !=
                | >=
                | <=
<Alter_For>:= ID ++;
                | ID -- ;
<While>:= while ( <Expresion> ) { <Sentencias> <Final_Sentencia> }
<Do_While>:= do { <Sentencias> <Final_Sentencia> } while ( <Expresion> ) ;
<Comentarios> := Tipo_Comentario

```