# PRACTICA 2

ORGANIZACIÓN DE LENGUAJES Y COMPILADORES 1

# MANUAL TECNICO

# TRADUCTOR C# A PYTHON

**FRANCISCO LUIS SUAREZ LÓPEZ**

**201807190**

# Desarrollo de la aplicación

# ANALISIS LEXICO

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

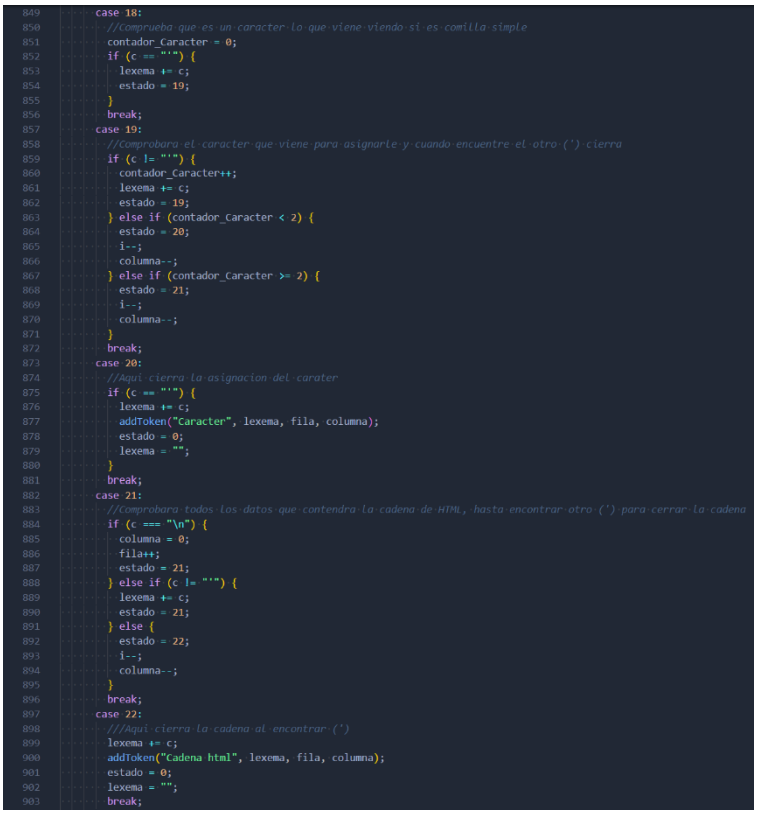
Letra . (Letra | Digito | _) *

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

(Digito)+ . (.)? . (Digito)+

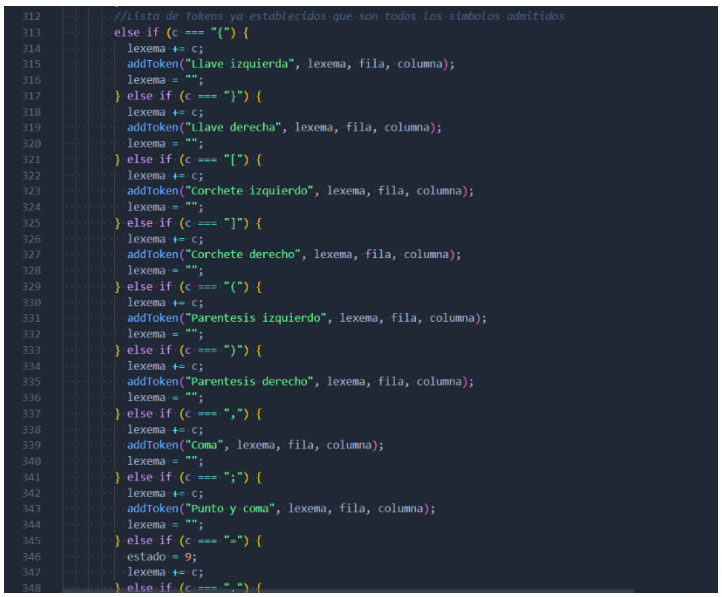
- Si se identifica el uso de comillas dobles, todo lo que venga después de eso se tomara como una cadena de texto y dejara de tomarlo hasta encontrar otras comillas dobles.

- 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.



```
case 18:
  //Comprueba que es un caracter lo que viene viendo si es comilla simple
  contador_Caracter = 0;
  if (c == "'") {
    lexema += c;
    estado = 19;
  }
  break;
case 19:
  //Comprobara el caracter que viene para asignarle y cuando encuentre el otro (') cierra
  if (c != "'") {
    contador_Caracter++;
    lexema += c;
    estado = 19;
  } else if (contador_Caracter < 2) {
    estado = 20;
    i--;
    columna--;
  } else if (contador_Caracter >= 2) {
    estado = 21;
    i--;
    columna--;
  }
  break;
case 20:
  //Aqui cierra la asignacion del caracter
  if (c == "'") {
    lexema += c;
    addToken("Caracter", lexema, fila, columna);
    estado = 0;
    lexema = "";
  }
  break;
case 21:
  //Comprobara todos los datos que contendra la cadena de HTML, hasta encontrar otro (') para cerrar la cadena
  if (c === "\n") {
    columna = 0;
    fila++;
    estado = 21;
  } else if (c != "'") {
    lexema += c;
    estado = 21;
  } else {
    estado = 22;
    i--;
    columna--;
  }
  break;
case 22:
  ///Aqui cierra la cadena al encontrar (')
  lexema += c;
  addToken("Cadena html", lexema, fila, columna);
  estado = 0;
  lexema = "";
  break;
```

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



```
//Lista de Tokens ya establecidos que son todos los simbolos admitidos
else if (c === "{") {
  lexema += c;
  addToken("Llave izquierda", lexema, fila, columna);
  lexema = "";
} else if (c === "}") {
  lexema += c;
  addToken("Llave derecha", lexema, fila, columna);
  lexema = "";
} else if (c === "[") {
  lexema += c;
  addToken("Corchete izquierdo", lexema, fila, columna);
  lexema = "";
} else if (c === "]") {
  lexema += c;
  addToken("Corchete derecho", lexema, fila, columna);
  lexema = "";
} else if (c === "(") {
  lexema += c;
  addToken("Parentesis izquierdo", lexema, fila, columna);
  lexema = "";
} else if (c === ")") {
  lexema += c;
  addToken("Parentesis derecho", lexema, fila, columna);
  lexema = "";
} else if (c === ",") {
  lexema += c;
  addToken("Coma", lexema, fila, columna);
  lexema = "";
} else if (c === ";") {
  lexema += c;
  addToken("Punto y coma", lexema, fila, columna);
  lexema = "";
} else if (c === "=") {
  estado = 9;
  lexema += c;
} 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.

```
default:
  lexema += c;
  if (
    c === "\n" ||
    c === "\\" ||
    c === "\t" ||
    c === " " ||
    c.charCodeAt(0) === 13 ||
    c.charCodeAt(0) === 9
  ) {
    estado = 0;
    lexema = "";
  } else {
    addError("Desconocido", lexema, fila, columna);
    estado = 0;
    lexema = "";
    pasoLibre = false;
  }
  break;
```

## ANALISIS SINTACTICO

```
//Funcion del parser
function parser() {
  //Vamos a añadir un ultimo token para saber donde termina
  addToken("Ultimo", "Ultimo", "0", "0");
  //Reiniciamos las listas
  ListaVariables = [];
  Lista_Errores_Sintaticos = [];
  //Reiniamos todos los valores
  indice = 0;
  tokenActual = Lista_de_Tokens[indice];
  errorSintactico = false;
  Error_Sintactico_Permiso = true;
  isVoid = false;
  //Codigo traducido python
  Contador_Tabs_Python = 0;
  Codigo_Python = "";
  //Llamada al no terminal inicial
  Inicio();
}
```

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.

```
function emparejar(tip) {
  if (tokenActual.Tipo != "Ultimo") {
    if (errorSintactico) {
      Error_Sintactico_Permiso = false;
      if (tokenActual.Tipo != "Ultimo") {
        console.log(tokenActual);
        indice++;
        tokenActual = Lista_de_Tokens[indice];
        if (
          tokenActual.Tipo === "Punto y coma" ||
          tokenActual.Tipo === "Llave derecha"
        ) {
          errorSintactico = false;
        }
      }
    } else {
      if (tokenActual.Tipo != "Ultimo") {
        if (tokenActual.Tipo === tip) {
          indice++;
          tokenActual = Lista_de_Tokens[indice];
        } else {
          addErrorSintactico(
            tip,
            tokenActual.Tipo,
            tokenActual.Fila,
            tokenActual.Columna
          );
          errorSintactico = true;
        }
      }
    }
  }
```

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 ( <Paramatros> ) { <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