12/6/2019

# Analizador Sintáctico

## Teoría Computacional

Isaac Hernández Salazar
UPSLP

# Introducción

En el siguiente documento se explicará el desarrollo de un analizador léxico-sintáctico para el lenguaje de programación interpretado PostgreSQL de la clase Teoría Computacional, fue programado en el lenguaje Delphi Pascal, el cual es un lenguaje de programación basado en objetos desarrollado por la empresa Embarcadero Technologies. Este lenguaje está totalmente basado en Pascal y su IDE, el cual se llama también Delphi, nos permite diseñar visualmente una aplicación nativa de Windows, tal como lo hace Visual Studio para .NET.

La solución a los problemas que presento el un analizador léxico-sintáctico se basó en análisis y planteamientos de autómatas, expresiones regulares y gramática de libre contexto, siendo esta ultima fundamental para el desarrollo del analizador sintáctico LL, el cual se explicara en el marco teórico.

# Marco Teórico

Un analizador léxico crea tokens de una secuencia de caracteres de entrada y son estos tokens los que son procesados por el analizador sintáctico para construir la estructura de datos. Un analizador sintáctico es un programa informático que analiza una cadena de símbolos de acuerdo con las reglas de una gramática de libre contexto.

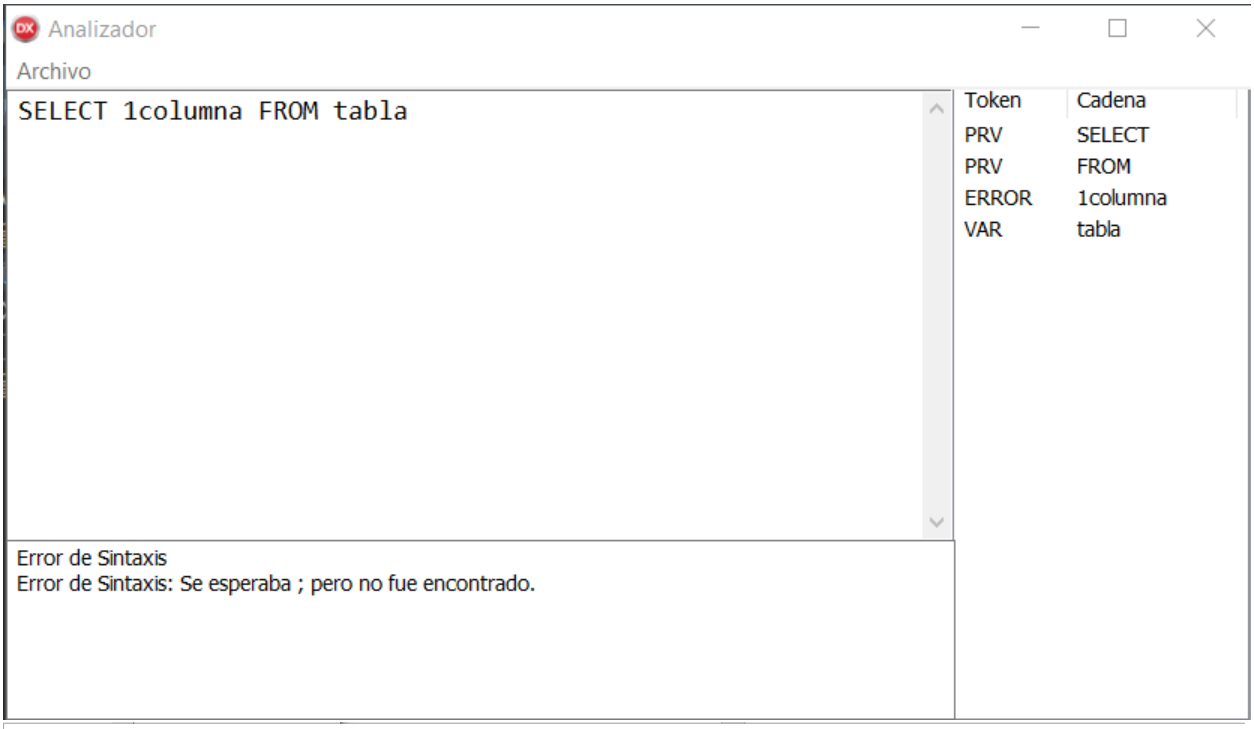
Las gramáticas libres de contexto permiten describir la mayoría de los lenguajes de programación, de hecho, la sintaxis de la mayoría de los lenguajes de programación está definida mediante gramáticas libres de contexto. Por otro lado, estas gramáticas son suficientemente simples como para permitir el diseño de eficientes algoritmos de análisis sintáctico que, para una cadena de caracteres dada, determinen cómo puede ser generada desde la gramática. Los analizadores LL y LR tratan restringidos subconjuntos de gramáticas libres de contexto.

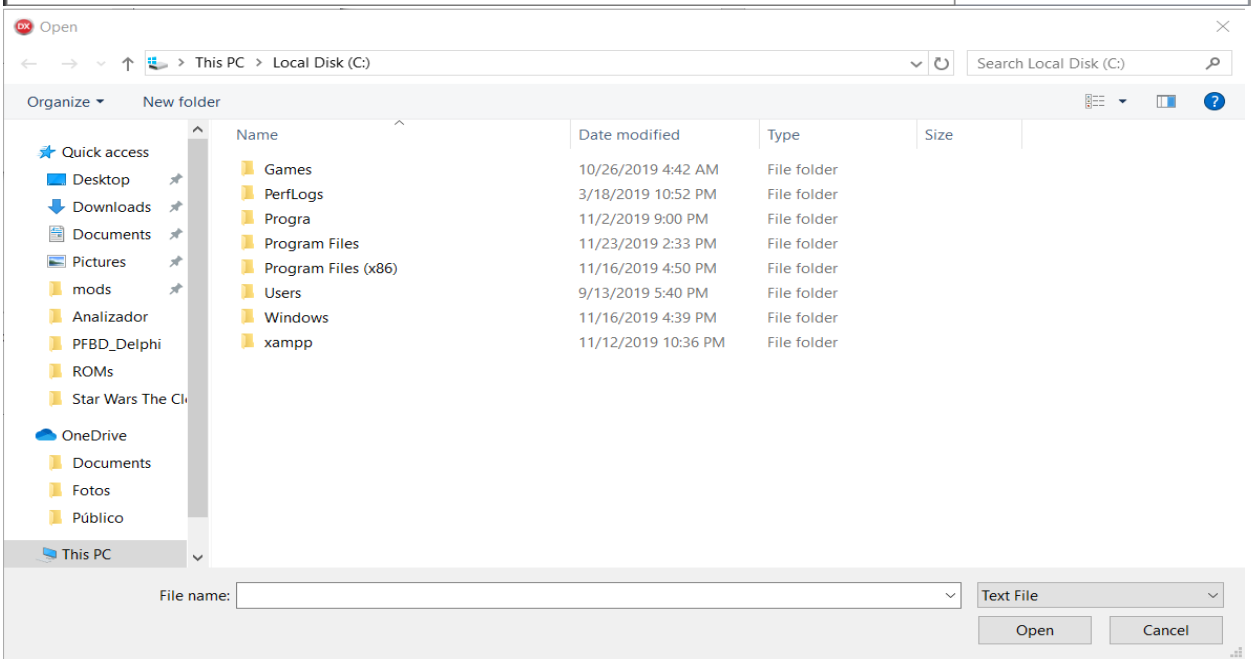
El analizador sintáctico LL es un analizador sintáctico descendente, por un conjunto de gramática libre de contexto. En este analizador las entradas son de izquierda a derecha, y construcciones de derivaciones por la izquierda de una sentencia o enunciado. La clase de gramática que es analizable por este método es conocido como gramática LL.

El autómata de pila es una extensión del autómata finito no determinista con transiciones-ε , el cual constituye una forma de definir los lenguajes regulares. El autómata a pila es fundamentalmente un AFN-ε con la adición de una pila. La pila se puede leer, se pueden introducir elementos en ella y extraer sólo el elemento que está en la parte superior de la misma, exactamente igual que la estructura de datos de una “pila”.

## Interfaz Gráfica de Usuario

El diseño de la interfaz grafica de usuario se baso en su mayor parte, de los IDEs de otros lenguajes de programación, tal como los son Eclipse o Code::Blocks, esta consta de un área para el código fuente, una lista de tokens en un costado, una barra de herramientas y en la parte inferior, un área donde se muestran mensajes generados por la aplicación, tal como lo son advertencias o errores. El código fuente puede ser escrito manualmente o cargado desde un archivo .txt o .sql, pese que un código fuente haya sido cargado desde un archivo, este puede seguir editándose manualmente. Una vez terminado el código fuente, podemos guardarlo como un .txt o .sql.



# Analizador Léxico

## Planteamiento

Como se menciono en el marco teórico, el analizador léxico se encarga de crear diferentes tokens a partir de una cadena de caracteres, los cuales servirán para ser procesados por el analizador sintáctico, por lo que se tendrá que definir los tipos de tokens que usaremos para el lenguaje PostgreSQL. Se definieron cinco tipos diferentes de tokens, que difieren entre ellos por su utilidad para el analizador sintáctico y su contexto dentro del lenguaje:

- Palabras reservadas
- Operadores lógicos y aritméticos
- Variables y constantes
- Otros símbolos
- Errores léxicos

La lógica de programación para esta parte del programa se enfocará en como diferenciar los tokens dentro del código fuente ingresado. A continuación, se explicarán las funciones, procedimientos y que fueron implementados para dar solución al analizador léxico.

## Definición del Léxico

Las palabras reservadas, operadores lógicos y aritméticos, y otros símbolos a utilizar ya son un léxico conocido de PostgreSQL, por lo que se utilizaran las mismas como el contenido de sus respectivos tokens. Se definieron un arreglo para cada uno de los tokens mencionados, que contienen todo el léxico que les corresponde.

```
const
  PALABRAS_RESERVADAS : Array[0..67] of String =
    (
      'SELECT', 'FROM', 'WHERE', 'ON',
      'IS', 'NOT', 'USING', 'ALL',
      'ALTER', 'AND', 'ANY', 'CASCADE',
      'INSERT', 'CONTAINS', 'CREATE', 'CROSS',
      'MAX', 'BY', 'CHECK', 'CURRENT_USER',
```

```
  OPERADORES : Array[0..10] of String =
    (
      '\x25', '\x26', '\x2A', '\x2B',
      '\x2D', '\x2F', '\x3C', '\x3D',
      '\x3E', '\x5E', '\x7C'
    );
  OTROS : Array[0..6] of String =
    (
      '\x3B', '\x28', '\x29', '\x5B', '\x5D', '\x2C', '\x27'
    );
```

Estos arreglos ayudaran a identificar las cadenas o caracteres correspondientes a los diferentes tokens dentro del codigo, fuente.

## Analizando el Codigo Fuente

Una vez teniendo el léxico de PostgreSQL definido en arreglos, se utilizaron varias funciones para extraer los tokens del código fuente, se utilizó una función por cada token. Las primeras cuatro funciones, analizan los tokens en base a varias expresiones regulares, mientras que la función para las variables y constantes se basa en desechar palabras que ya contenían otros tokens, considerando a las permanecientes como tokens de variables.

```
type
  AnalizadorLexico = Class(TObject)
    private
      function RestarArreglos(A1, A2: TArray<String>): TArray<String>;
    public
      function AnalizarPRV(textbox: String): TArray<String>;
      function AnalizarOLA(textbox: String): TArray<String>;
      function AnalizarOTROS(textbox: String): TArray<String>;
      function AnalizarERROR(textbox: String): TArray<String>;
      function AnalizarVAR(textbox: String; PRV, OLA, OTROS, ERROR: TArray<String>): TArray<String>;
    end;
```

Las cuatro funciones ajenas a las variables funcionan exactamente de la misma manera, haciendo uso de sus respectivos arreglos, dada una expresión regular extraen todas las cadenas que coinciden con esta misma, insertándolas dentro de un arreglo que será lo que retorne cada una de las funciones.

```
function AnalizadorLexico.AnalizarPRV(textbox: String): TArray<String>;
var
  regex: TRegEx;
  match: TMatchCollection;
  i, j, k: Integer;
begin
  k := 1;
  for i := 0 to Length(PALABRAS_RESERVADAS)-1 do
  begin
    regex.Create('\b'+PALABRAS_RESERVADAS[i]+'\b');
    match := regex.Matches(textbox);
    for j := 0 to match.Count-1 do
    begin
      SetLength(Result, k);
      Result[k-1] := match.Item[j].Value;
      k := k + 1;
    end;
  end;
end;
```

Los tokens de errores son los únicos que no dependen de un arreglo estos se basan en una de las pocas reglas léxicas de PostgreSQL, la cual dice que “El nombre de una variable no puede comenzar con un número, solo puede llevarlos después de una letra o un guion bajo”. La regla mencionada corresponde a una variable, por lo que la regla para un error seria “Una palabra del token error es aquella que comienza con un número y contiene por lo menos una letra o guion bajo”.

```
  regex.Create('\b\d+[a-zA-Z_]+\b');
  match := regex.Matches(textbox);
  for j := 0 to match.Count-1 do
```

### Imprimiendo Resultados del Análisis

Una vez obtenidos los tokens del código fuente, es necesario mostrarlos en pantalla, para esto se usó un TListView, el cual es una tabla a la cual agregaremos, la cadena encontrada y el token correspondiente a esa cadena y con esto concluiría el analizador léxico del programa.

```
procedure TAForm.SetTableTokens(PRV, OLA, OTROS, ERROR, TVAR: TArray<String>);
var
  i: Integer;
begin
  with TokenList do
  begin
    Parent := Panel1;
    Align := alClient;
    ViewStyle := vsReport;
    Items.Clear;
    for i := 0 to Length(PRV)-1 do
    begin
      ListItem := Items.Add;
      ListItem.Caption := 'PRV';
      ListItem.SubItems.Add(PRV[i]);
```

| Token | Cadena |
|---|---|
| PRV | SELECT |
| PRV | FROM |
| OLA | > |
| OTROS | ; |
| ERROR | 1d_ |
| VAR | casa |

## Analizador Sintáctico

### Planteamiento

Al igual que en el español, se necesitan reglas para formar frases usando palabras, en el caso de un lenguaje de programación, se necesitan reglas para escribir sentencias o líneas de código usando el léxico del lenguaje. El analizador se centrará en la definición de la sintaxis de la sentencia SELECT, la cual es la más conocida de SQL en general, teniendo LIMIT y OFFSET como diferencias notables del SELECT de PostgreSQL con respecto a otros lenguajes del Estándar SQL. A continuación, se muestra la sintaxis de SELECT extraída de la documentación de PostgreSQL.

```
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    * | expression [ [ AS ] output_name ] [, ...]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
```

La sintaxis mostrada se utilizo como base para la creación de la gramática de libre contexto, de la sentencia en sí, y otros elementos que se incluyen, tales como, condiciones, variables y parámetros.

Para reducir la complejidad de la sintaxis en este proyecto, se eliminaron elementos como WINDOW, ALL y ASC, facilitando así, la sintaxis de la sentencia SELECT.

En el siguiente punto se vera la sintaxis nueva que se utilizara en la codificación.

## Gramática de Libre Contexto

Para definir una gramática de libre contexto se pensó en los tokens variables y no variables dentro de la sentencia SELECT, para el caso de este programa, se dejo el FROM o el INTO como obligatorios. A continuación, se muestra la gramática definida para el analizador sintáctico.

SELECT <Expresiones> <Ruta> ;

<Expresiones> -> *
<Valores>

<Valores> -> Números
Palabras
<Valores>, <Valores>

<Ruta> -> FROM <Variables> <Opcional>
INTO <Variables>

<Variables> -> Palabras
<Variables>, <Variables>

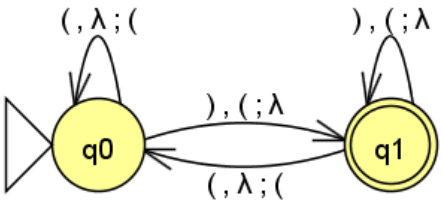
<Opcional> -> Empty
WHERE <Condición> <Opcional> //Opcional sin WHERE
USING ( Palabra ) <Opcional> //Opcional sin USING y anteriores
GROUP BY <Valores> <Opcional> //Opcional sin GROUP BY y anteriores
HAVING <Condición> <Opcional> //Opcional sin HAVING y anteriores
LIMIT Números <Opcional> //Opcional sin LIMIT y anteriores
OFFSET Números

<Condición> -> Palabra = | > | < Palabra <Extra_Condición>

<Extra_Condición> -> Empty
AND | OR <Condición>

## Autómata de Pila

En la gramática de libre contexto definida anteriormente, se observa que la función USING requiere que el valor usado este encerrado por paréntesis, para validar que un paréntesis abierto posteriormente haya sido cerrado, se uso un autómata de pila. Este no solo es para paréntesis, si no también, para corchetes y las palabras BEGIN y END. Metiendo la respectiva cadena a una pila y sacando la punta de la pila cuando su opuesto haya sido encontrado, si encuentra un cierre que no corresponde al suyo, mandara error instantáneamente, en caso de terminar de analizar, verificara que la pila se encuentre vacía. La siguiente imagen muestra el autómata de pila tomado como referencia para su codificación mediante recursividad.



Ambas funciones del autómata de pila se programaron tal y como se mencionó, se analiza el código fuente de forma secuencial, cuando se encuentra un símbolo de apertura, este se agrega a la pila, y cuando se encuentra un símbolo de cerradura su contraparte es eliminada, en caso de no ser la contra parte quien esta en la cima de la pila, se mandará un error, ya que se encontró otro símbolo. Esto ultimo pasara de igual manera si se intenta sacar un elemento de la pila y esta se encuentra vacía.

```
procedure AnalizadorSintactico.q0(pos, textbox: String);
var
  x, y: char;
begin
  Match := Match.NextMatch();
  if not(pos.IsEmpty) then x := pos.Chars[0];
  case x of
   '(','[','B': begin
                  Stack.Push(x);
                  if Match.Success then q0(Match.Value, textbox);
                end;
  end;
  case x of
   ')' : y := '(';
   ']' : y := '[';
   'E' : y := 'B';
  end;
  case x of
   ')',']','E': begin
                  if (Stack.Count <> 0) and (Stack.Peek.Equals(y)) then Stack.Extract() else qx(x);
                  if Match.Success then q1(Match.Value, textbox);
                end;
  end;
end;
```

```
procedure AnalizadorSintactico.qx(x: String);
var
  y: String;
begin
  Flag := False;
  if Stack.Count <> 0 then
  case Stack.Peek.Chars[0] of
   '(' : y := ')';
   '[' : y := ']';
   'B' : y := 'END';
  end;
  if x.Equals('E') then x:= 'END';
  if not y.IsEmpty then Errors := Errors + SLineBreak + 'Error de Sintaxis: Se e
  else Errors := Errors + SLineBreak + 'Error de Sintaxis: No se esperaba '+x+'
end;
```

## Gramática de Libre Contexto a Código

La tarea mas extensa del analizador sintáctico es codificar las reglas descritas en nuestra gramática de libre contexto. En la gramática definida se puede observar que existe recursividad, por ejemplo, en el caso de las variables, para evitar programar demasiada recursividad se utilizaron expresiones regulares con cerradura positiva y cerradura de Kleene, mientras que las reglas con posibilidad de ser vacías se implementaron como condiciones no obligatorias, pero que en caso de existir deben cumplir las reglas descritas en la gramática.

La sintaxis opcional de la sentencia SELECT no presento problemas ya que, al llamar las funciones de manera secuencial, al revisar una regla posterior como LIMIT, no habría posibilidad de retornar a las reglas anteriores, por lo que, al no seguir el camino definido, la sintaxis seria marcada como incorrecta.

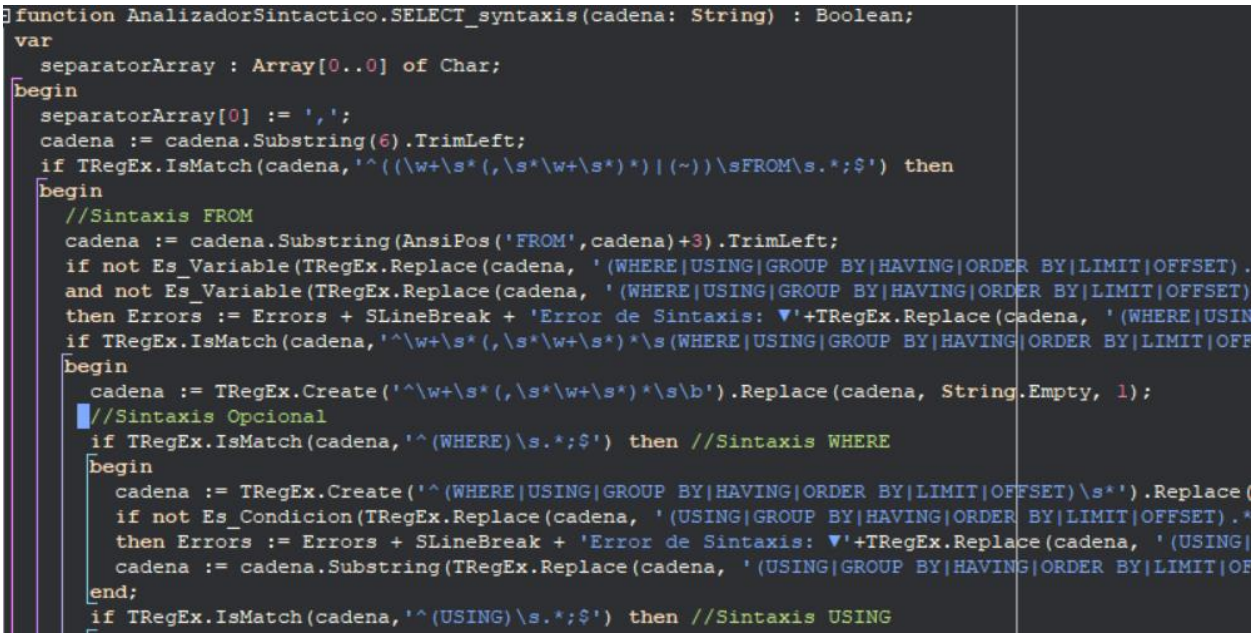
La clase del analizador sintáctico solo cuenta con una sobrecarga de método en la función Es_Variable, que como su nombre lo dice, valida si algo recibido es variable o no. Al tratarse de varias variables separadas por comas el método es capaz de recibir un arreglo de cadenas de todas las palabras que se desean validar, en caso de que al menos una no sea válida, todo el resultado será falso.

```
function AnalizadorSintactico.Es_Variable(cadena: String) : Boolean;
var
  i: Integer;
begin
  if not(TRegEx.IsMatch(cadena, '^[a-zA-Z_]+(\d+[a-zA-Z_]*)*$')) or TArray.BinarySearch<String>(PR,
  else Result := True;
end;

function AnalizadorSintactico.Es_Variable(arreglo: TArray<String>) : Boolean;
var
  i: Integer;
  cadena: String;
begin
  Result := True;
  for cadena in arreglo do
  begin
    if not(TRegEx.IsMatch(cadena, '^[a-zA-Z_]+(\d+[a-zA-Z_]*)*$')) or TArray.BinarySearch<String>(P
  end;
end;
```

Para facilitar el análisis sintáctico, el programa cuenta con dos funciones, Sentencias y Funciones, las cuales sirven para separar e idéntica las sentencias respectivamente.

```
procedure AnalizadorSintactico.Sentencias(cadena: String);
begin
  if cadena.Chars[cadena.Length-1] <> ';' then
  begin
    Flag := False;
    Errors := Errors + SLineBreak + 'Error de Sintaxis: Se esperaba ; pero no fu
  end;
end;

procedure AnalizadorSintactico.Funciones(cadena: String);
begin
  cadena := cadena.TrimLeft;
  cadena := cadena.TrimRight;
  if (TRegEx.IsMatch(cadena, '^SELECT\s.*;$')) then
  begin
    SELECT_syntaxis(cadena);
  end;
  Match := Match.NextMatch();
  if Match.Success then Funciones(Match.Value);
end;
```

Una vez identificada y asilada la sentencia, se procederá a validar su sintaxis, mediante el recorrido principal de la gramática de libre contexto, haciendo uso de las funciones previamente definidas para validar reglas de otros niveles, como son las condiciones, variables y asignaciones. De igual manera, se comenzó a analizar las sentencias de izquierda a derecha, separando y validando los elementos ajenos a las palabras reservadas definidas en la sintaxis.



```
function AnalizadorSintactico.SELECT_syntaxis(cadena: String) : Boolean;
var
  separatorArray : Array[0..0] of Char;
begin
  separatorArray[0] := ',';
  cadena := cadena.Substring(6).TrimLeft;
  if TRegEx.IsMatch(cadena,'^((\w+\s*(,\s*\w+\s*)*)|(~))\sFROM\s.*;$') then
  begin
    //Sintaxis FROM
    cadena := cadena.Substring(AnsiPos('FROM',cadena)+3).TrimLeft;
    if not Es_Variable(TRegEx.Replace(cadena, '(WHERE|USING|GROUP BY|HAVING|ORDER BY|LIMIT|OFFSET).
    and not Es_Variable(TRegEx.Replace(cadena, '(WHERE|USING|GROUP BY|HAVING|ORDER BY|LIMIT|OFFSET)
    then Errors := Errors + SLineBreak + 'Error de Sintaxis: ▼'+TRegEx.Replace(cadena, '(WHERE|USIN
    if TRegEx.IsMatch(cadena,'^\w+\s*(,\s*\w+\s*)*\s(WHERE|USING|GROUP BY|HAVING|ORDER BY|LIMIT|OFF
    begin
      cadena := TRegEx.Create('^\w+\s*(,\s*\w+\s*)*\s\b').Replace(cadena, String.Empty, 1);
      //Sintaxis Opcional
      if TRegEx.IsMatch(cadena,'^(WHERE)\s.*;$') then //Sintaxis WHERE
      begin
        cadena := TRegEx.Create('^(WHERE|USING|GROUP BY|HAVING|ORDER BY|LIMIT|OFFSET)\s*').Replace(
        if not Es_Condicion(TRegEx.Replace(cadena, '(USING|GROUP BY|HAVING|ORDER BY|LIMIT|OFFSET).*
        then Errors := Errors + SLineBreak + 'Error de Sintaxis: ▼'+TRegEx.Replace(cadena, '(USING|
        cadena := cadena.Substring(TRegEx.Replace(cadena, '(USING|GROUP BY|HAVING|ORDER BY|LIMIT|OF
      end;
      if TRegEx.IsMatch(cadena,'^(USING)\s.*;$') then //Sintaxis USING
```

## Conclusión

Este proyecto fue de gran utilidad para la comprensión de temas de la materia de Teoría Computacional, en especial, la gramática de libre contexto y como pasar de una gramática de libre contexto a un autómata. A su vez, se reforzaron temas como la recursividad, la programación orientada a objetos y el manejo de cadenas.

Pese a no ser un analizador sintáctico tan completo como lo es, el que se usa en el gestor de bases de datos PostgreSQL, ayudo a entender las bases y el funcionamiento de este tipo de programas.

## Referencias


Group, T. P. (3 de Diciembre de 2019). *PostgreSQL 10.11 Documentation*. Obtenido de PostgreSQL: https://www.postgresql.org/docs/10/index.html

Hopcroft, J. E., Motwani, R., & Ullman, J. D. (2007). *Introducción a la teoría de autómatas, lenguajes y computación.* Madrid: PEARSON EDUCACIÓN S.A.