MANUAL TECNICO

Este manual contiene la explicación de las técnicas que se utilizar para solventar la problemática del análisis de un archivo de entrada.cs el cual corresponde a un lenguaje llamado C#.

A continuación, se explica de forma explícita los métodos utilizados y ciertas partes del código;

para el análisis léxico se analizó la entrada carácter por carácter, concatenando cuando era necesario (variables auxiliares léxica) y preguntando si pertenece o no al lenguaje, si pertenece este pasa a formar un Token de lo contrario sería un error léxico el cual se adjunta a un vector (los vectores no tienen límites en javascript ni en typescript)

```
public analisis_lexico(cadena_entrada:string ){
    console.log("ANALIZANDO: " + cadena_entrada);
    cadena_entrada += "\n";
    this.estado = 0; // inicia en el estado de inicio
    this.aux_lexico = ""; // cadena acumuladora de lexema acutual
    let act:string =""; // caracter actual  o tipo string

    for (let i = 0; i < cadena_entrada.length; i++) {
        Estatico.COLUMNAS++;
        act = cadena_entrada[i];

        switch (this.estado)
        {
            case 0:
            if (act == "\n")
            {
                Estatico.FILAS++;
                Estatico.COLUMNAS = 0 ;
            }else if (act == " " || act == "\t")
            {
                // solo no lo reconoce como un error
            }else if (act =="+"){ this.aux_lexico += act; this.estado = 5; }
            else if (act =="-") {this.aux_lexico += act; this.estado = 7; }
            else if (act =="*") { this.aux_lexico += act; this.addToken(Tipo.sb_por);}
            else if (act =="/") { this.aux_lexico += act; this.estado = 8; }
            else if (act =="(") { this.aux_lexico += act; this.addToken(Tipo.parentesis_izq);}
            else if (act ==")")  { this.aux_lexico += act; this.addToken(Tipo.parentesis_derecho); }
            else if (act =="{")  { this.aux_lexico += act; this.addToken(Tipo.llave_izq); }
            else if (act =="}")  { this.aux_lexico += act; this.addToken(Tipo.llave_derecha); }
            else if (act =="=")  { this.aux_lexico += act; this.estado = 6; }
            else if (act ==":")  { this.aux_lexico += act; this.addToken(Tipo.dosPuntos); }
```

Como se muestra en la imagen solo se va iterando por medio de un ciclo para poder pasar por cada carácter de la cadena de entrada.

Para en análisis Sintáctico se utilizó una gramática descendente y se aplicó el método de modo pánico, dicho método modo pánico consiste en que si hay un error lo muestra y todos los demás tokens los descarta hasta encontrar alguno que simbolice la finalización de una sentencia en el caso del proyecto esos tokens de finalización y recuperación eran el punto y la llave de cierre.

La gramática utilizada fue la siguiente:

Inicio -> sentencia_clase lista_clasesP "Sharp"

lista_clasesP -> sentencia_clase lista_clasesP

| epsilon

sentencia_clase -> "P_Class" "ID" "{" Lista_Declaraciones_metFunVar "}"

Lista_Declaraciones_metFunVar-> Declaracion Lista_Declaraciones_metFunVarP

|epsilon

Lista_Declaraciones_metFunVarP-> Declaracion Lista_Declaraciones_metFunVarP

| epsilon

Lista_inst -> Instruccion Lista_instP

Lista_instP-> Instruccion Lista_instP

|epsilon

Instruccion-> DECLARACION_ADENTRO_DE_METODOS_FUNCIONES

|Sentencia_while

|Sentencia_for

|SentenciaImprime

|Sentencia_if

|SentenciaSwitch_case

|asignacionSimple

|Sentencia_do_while

|Sentencia_return_funciones

|Sentencia_continue

|sentencia_break

|epsilon

ListaIns_entreLLaves-> "{" Lista_inst "}"

opcionMetodoFuncion-> Tipo “ID” lista_parametros ")" "{" Lista_inst "}"

| ")" "{" Lista_inst "}"

lista_parametros -> "," Tipo “ID” lista_parametros

| epsilon

DECLARACION_ADENTRO_DE_METODOS_FUNCIONES-> tipo “ID” DeclaracionP_metodos

DeclaracionP_metodos -> Lista_ids asignacion ";"

Declaracion->” p_res_void” “ ID” "(" opcionMetodoFuncion

| tipo ID DeclaracionP

DeclaracionP ->"(" opcionMetodoFuncion

|Lista_ids asignacion ";"

Lista_ids-> "," “ID “ lista_ids

| epsilon

asignacion'-> "=" expresion

| epsilon

asignacionSimple -> ID OpcionAsignacion

OpcionAsignacion-> "=" expresion ";"

| "(" sentencia_llama_metodo ";"

sentencia_break-> p_break ";"

Sentencia_do_while-> “p_res_do” ListaIns_entreLLaves “ palabra_while “ "(" lista_expresiones_condicionales ")" ";"

sentencia_continue->” p_res_continue “ ";"

Sentencia_return_funciones-> “p_res_return” expresion ";"

Sentencia_return_metodos-> “ p_res_return” ";"

ListaExpresiones -> expresion Lista_expresionP

Lista_expresionP -> "," expresion Lista_expresionP

|epsilon

Sentencia_if -> P_if "(" lista_expresiones_condicionales ")" ListaIns_entreLLaves else'

else'-> “Palabra_else” multiplesIf

|epsilon

multiplesIf -> ListaIns_entreLLaves

| sentencia_if

Sentencia_while -> “P_while” "(" lista_expresiones_condicionales ")" ListaIns_entreLLaves

Sentencia_for ->” P_for” "(" declaracionFOR ";" lista_expresiones_condicionales ";" “ID” DecrementoIncremento ")" ListaIns_entreLLaves

DecrementoIncremento -> "++"

|"--"

declaracionFOR -> Tipo “ID” "=" expresion

| “ID” "=" expresion

lista_expresiones_condicionales -> expresion ListaExpresionesCondicionalesP

ListaExpresionesCondicionalesP -> "&&" expresion ListaExpresionesCondicionalesP

| "||" expresion ListaExpresionesCondicionalesP

| epsilon

Tipo -> Int

|double

|Char

|String

|Bool

SentenciaImprime -> “P_res_Console” ". " P_WriteLine "(" expresion' ")" ";"

SentenciaSwitch_case ->” P_switch “ "(" “ID” ")" "{" ListaCases Default' "}"

ListaCases -> case listaCase'

listaCase' -> case listaCase'

|epsilon

caseP ->p_case opcionCase ":" Lista_inst sentencia_break

OpcionBreak-> “p_break” ";"

| epsilon

OpcionCase-> “Num”

| “cadena”

| “caracter”

| “bool”

Default' -> P_default ":" Lista_inst p_break ";"

|epsilon

Expresion->E simboloComparacionOpcional

simboloComparacionOpcional -> == E

| > E

| < E

| <= E

| >= E

| != E

| Epsilon

E-> TE'

E'->+TE'

|-TE'

| Epsilon

T-> FT'

T'->*FT'

| / FT'

|Epsilon

F-> Decimales

| Cadena

| ID ExpresionMetodo

| true

| false

| Numero

| caracter

| !E

| (Expresion)

ExpresionMetodo -> "(" sentencia_llama_metodo

| epsilon

sentencia_llama_metodo -> ListaExpresiones ")"

| ")"

sentencia_llama_metodo -> Tipo “ID” lista_parametros ")"

| ")"

sentencia_llama_metodo -> ")"

| ListaExpresiones ")"

En código cada producción (letras en negrilla) representa un método y cada NO terminal es una llamada a un método y los terminales es una llamada a un método que los hace match en este caso ese método se llama parea.

Este método sin duda alguna es el más importante de la clase ya que determina si viene algo no esperado y de ser así un error sintáctico, este avanza de token es el encargado de ir moviendo cada token cuando se debe.

En este caso se puede apreciar la codificación de una producción la cual puede estar compuesta por terminales y no terminales, teniendo en cuenta que esta forma de implementar la gramática descendente es si o si recursiva y a su vez predictiva ya que primero tenemos que preguntar qué camino tomar para saber a qué producción ir y así ejecutar correctamente el análisis.