Grupo 25

Práctica PDL

ETSIINF 2020-21

Autores:

Guillermo García Pradales

Floriana Antonia Pavel

Daniel Rodríguez Mariblanca

Contenido

1. Introducción

El objetivo principal de esta práctica es desarrollar el diseño e implementación de un procesador de lenguajes para una versión alternativa de JavaScript. Para ello, se han de realizar los análisis léxicos, sintácticos y semánticos, además de incluir la tabla de símbolos y un gestor de errores.

Así, a lo largo de este documento se explicarán las decisiones tomadas para el diseño de los analizadores y para su implementación correcta, realizada en el lenguaje Python. Se ha elegido implementar todo el procesador del lenguaje en Python debido a su versatilidad y facilidad de sintaxis.

2. Diseño del Analizador Léxico

Tokens

< ID, PunteroTablaSimbolos >	< NEGATION, >
< PLUS, >	< ALERT, >
< MINUS, >	< BOOLEAN, >
< LPARENT, >	< ELSE, >
< RPARENT, >	< FUNCTION, >
< ASSIGN, >	< IF, >
< COMMA, >	< INPUT, >
< SEMICOLON, >	< LET, >
< LESST, >	< NUMBER, >
< GREATERT, >	< RETURN, >
< REMAINDER, >	< STRING, >
< CHAIN, lexema>	< CONSTNUM, valor >

Gramática

```
l = {a-z ,A-Z, _ }  // Es decir, todas las letras incluyendo la barra baja (_)
c = {cualquier carácter}
c´= {cualquier carácter menos * (asterisco)}
c´´= {cualquier carácter menos / (barra)}
c´´´ = {cualquier carácter menos ‘ (comilla simple)}
```

d = {0-9}

S → delS | lA | dC | %D | ‘E | + | - | (|) | = | , | ; | < | > | ! | /F

A → lA | dB | λ

B → dB | lA | λ

C → dC | λ

D → =

E → c´´´D | ‘

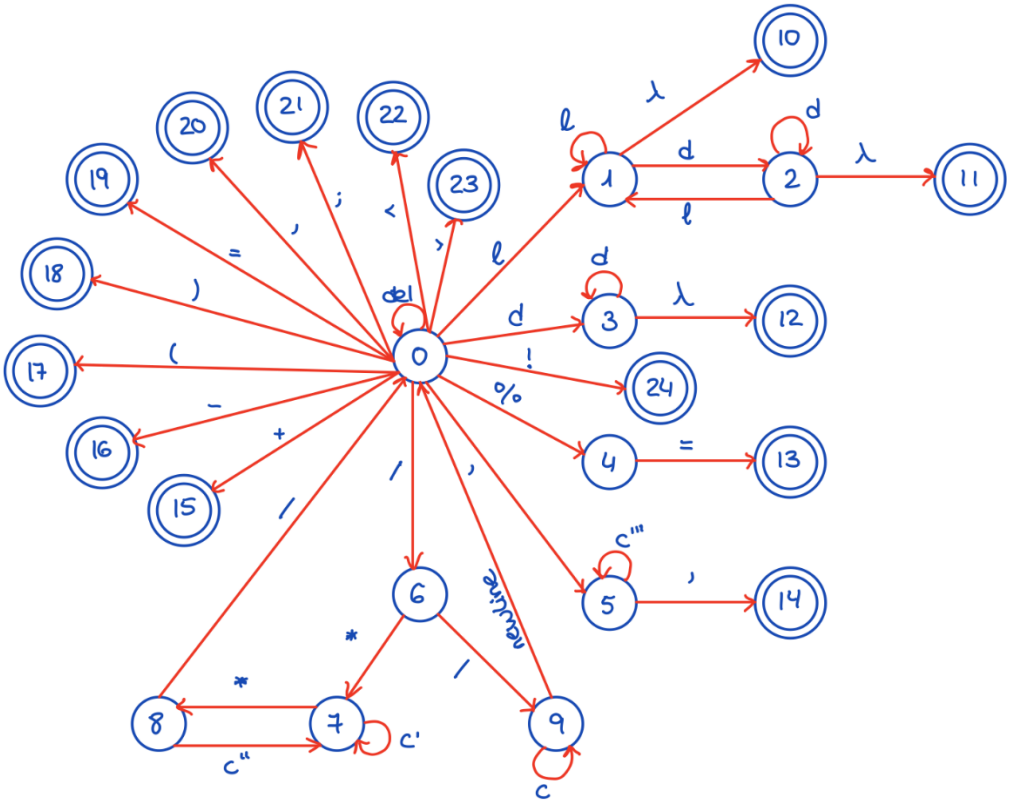
F → *G | /I

G → c´G | *H

H → c´´G | /

I → cI | newLine

Autómata



Acciones semánticas y errores

En todas las acciones semánticas, se lee y las acciones que no estén especificadas aquí serán errores:

```
0-0: Leer

0-1: lexema := l

0-3: numero := d

0-4: lexema := %

0-5: lexema := ϕ

1-1, 2-1: lexema := lexema + l

1-2, 2-2: lexema := lexema + d

1-10:  palres := buscarPalabraReservada(lexema)
             If (palres = null) Then
                    If Zona_Declaración = true Then
                           p := buscarTS(lexema)
                           If p = null then p := insertar_TS(lexema),
      GenToken(lexema, p)
                           Else error("Identificador ya declarado")
                    Else /* Zona_Declaración = false */
                           P := buscarTS(lexema)
                           If p = null Then error("Identificador no declarado")
                           Else GenToken(lexema, p) /* p != null */
             Else /* palres != null */
                    If Zona_Declaración = true Then
                           error ("Un identificador no puede llamarse como una
                           palabra reservada")
                    Else if (lexema = alert) Then GenToken (ALERT, )
                    Else if (lexema = input) Then GenToken (INPUT, )
                    Else if (lexema = boolean) Then GenToken (BOOLEAN, )
                    Else if (lexema = let) Then GenToken (LET, )
                    Else if (lexema = number) Then GenToken (NUMBER, )
                    Else if (lexema = string) Then GenToken (STRING, )
                    Else if (lexema = if) Then GenToken (IF, )
                    Else if (lexema = else) Then GenToken (ELSE, )
                    Else if (lexema = function) Then GenToken (FUNCTION, )
                    Else GenToken (RETURN, ) /* lexema = return */

2-11:  If Zona_Declaración = true Then
```

```
            p := buscarTS(lexema)
            If p = null then p := insertar_TS(lexema), GenToken(lexema, p)
            Else error("Identificador ya declarado")
      Else /* Zona_Declaración = false */
            P := buscarTS(lexema)
            If p = null Then error("Identificador no declarado")
            Else GenToken(lexema, p) /* p != null */
3-3: numero := numero * 10 + valorA(d)
3-12:  If (numero >= 65 536) Then
            error("El número + numero + se sale del rango [0-65536].")
      Else GenToken(CONSTNUM, numero)
4-13: GenToken(REMAINDER, )
5-5: lexema := lexema + c'''
5-14: GenToken(CHAIN, lexema)
0-6, 6-7, 6-9, 7-8, 8-0, 9-0: Leer
0-15:       GenToken(PLUS, )
0-16:  GenToken(MINUS, )
0-17:  GenToken(LPARENT, )
0-18:  GenToken(RPARENT, )
0-19:  GenToken(ASSIGN, )
0-20:  GenToken(COMMA, )
0-21:  GenToken(SEMICOLON, )
0-22:  GenToken(LESST, )
0-23:  GenToken(GREATERT, )
0-24:  GenToken(NEGATION, )
```

Diseño inicial de la tabla de símbolos

La estructura almacena toda la información relevante sobre los identificadores del programa:

- "index": indica la posición que ocupa en la tabla de símbolos
- "lexema": nombre del identificador
- "Tipo": indica el tipo del identificador (entero, array…)
- "Despl": dirección relativa
- "numParam": número de parámetros que tiene la función
- "TipoParamXX": tipos de parámetros que tiene la función. XX representa un número de dos dígitos, cuyos valores irán desde el 1 hasta el valor del atributo numParam.
- "TipoRetorno": tipo devuelto por la función
- "EtiqFuncion": etiqueta de comienzo del código de la función
- **"**Param**"**: representa que un identificador local es un parámetro formal del subprograma donde está declarado.

Se tiene una entrada por cada identificador. El método de organización es lineal; es decir, según llegan los tokens a la tabla de símbolos, se van colocando uno tras otro. El tamaño de la tabla es dinámico, pues no preestablecemos el tamaño.

Para más adelante, se va a manejar una tabla de símbolos por cada ámbito.

Como ya se ha mencionado, el lenguaje empleado para implementar la práctica es Python. En consecuencia, la estructura de datos escogida para almacenar los datos de la tabla de símbolos es un diccionario. El cual usa como llave el lexema de los identificadores. Y cada entrada en la tabla es otro diccionario que contiene como llave los elementos mencionados anteriormente.

3. Diseño del Analizador Sintáctico

Gramática

Las opciones del grupo 25 para la práctica son las siguientes:

- Sentencias: sentencia condicional compuesta (if, if-else)
- Operadores especiales: asignación con resto (%=)
- Técnicas de Análisis Sintáctico: ascendente
- Comentarios: comentario de bloque (/* */)
- Cadenas: con comillas simples (‘ ‘).

La gramática libre de contexto del analizador sintáctico es la siguiente:

G = (NoTerminales, Terminales, Produccciones, Axioma)

Axioma = Axioma_

NoTerminales = { Axioma_ Axioma Sentencia Senten Lista_Sentencias Funcion F1 F2 F3 Cabecera Parametros K K2 IF_ IF1 IF2 S X E R U V Tipo Tipo_B }

Terminales = { id ent cadena + - ! = %= ( ) { } , ; < > if else function alert input number let return string boolean }

Producciones = {

Axioma_ -> Axioma

Axioma -> Sentencia Axioma

Axioma -> Funcion Axioma

Axioma -> lambda

Sentencia -> S

Sentencia -> IF_

Sentencia -> let Tipo id ;

IF_ -> IF1 IF2

IF1 -> if (E) Senten

IF2 -> else Senten

IF2 -> lambd

Senten -> Sentencia

Senten -> { Lista_Sentencias }

S -> id = E ;

S -> id %= E ;

S -> id (Parametros) ;

S -> alert (E) ;

S -> input (id) ;

S -> return X ;

Parametros -> E K2

Parametros -> lambda

K2 -> , E K2

K2 -> lambda

Parametros -> E K2

Parametros -> lambda

K2 -> , E K2

K2 -> lambda

X -> E

X -> lambda

Funcion -> F1 F2 F3

F1 -> function Tipo_B id

F2 -> (Cabecera)

F3 -> { Lista_Sentencias }

Tipo -> boolean

Tipo -> number

Tipo -> string

Tipo_B -> Tipo

Tipo_B -> lambda

Cabecera -> Tipo id K

Cabecera -> lambda

K -> , Tipo id K

K -> lambda

Lista_Sentencias -> Sentencia Lista_Sentencias

Lista_Sentencias -> lambda

E -> ! V

E -> R

R -> R > U

R -> R < U

R -> U

U -> U + V

U -> U - V

U -> V

V -> id

Demostración

Nuestro analizador es sintáctico ascendente, que construye el árbol desde las hojas hacia la raíz. Su funcionamiento se basa en la reducción-desplazamiento; es decir, que siempre que se pueda, se aplica una regla (reducir) y en caso contrario se desplaza a su pila de trabajo el token que le está pasando el Analizados Léxico.

Este analizador se puede construir de forma eficiente usando la gramática LR. Se hace uso de una pila auxiliar de trabajo, en la que va generando de forma implícita el árbol de derivación, y cuenta con una tabla de análisis la cual se compone por la parte "acción" y la parte "goto".

Esta gramática ha de ser LR para poder ser analizada por el analizador sintáctico ascendente. Para ello, la forma más eficiente de hacerlo para una gramática de las dimensiones de la nuestra es realizando la tabla LR para asegurarse de que no hay dos posibles combinaciones dentro de una misma celda, esto es, no hay conflicto reducción/reducción o reducción/desplazamiento. A la vez, se confirma que no hay ninguna columna y fila vacía.

Tabla LR

Realizando la tabla LR con nuestra herramienta automática se obtiene una tabla en la que no hay conflictos y; por tanto, nuestra gramática es LR y apta para el analizador sintáctico. Las cuatro posibles acciones con las que nos podemos encontrar son las siguientes: desplazar, reducir, aceptar o error.

Debido a la complejidad de la gramática y el elevado número de estados, hemos decidido facilitar el siguiente enlace (I) para una mejor visualización del contenido que se nos pide. Las tablas han sido elaboradas en la siguiente página web (II).

(I) Link en GitHub de los FIRST, FOLLOWS y la Tabla LR(1)
(II) Generador SLR(1)

Autómata

Por el mismo motivo que en el apartado anterior, se ha decidido facilitar un enlace de visualización de los estados del autómata.

Link del Autómata

Deberá incluirse el listado del código fuente de cada caso de prueba, el volcado de los ficheros de parse y el árbol sintáctico generado con la herramienta VASt (para los 3 casos de prueba correctos) o el listado de errores detectados (para los 3 casos de prueba incorrectos).
Con respecto a las pruebas del sintáctico, para una visualización más completa, especialmente en relación con el árbol sintáctico generado con la herramienta VASt, se vuelve a facilitar un enlace que recopila toda la información deseada: el listado del código fuente de cada caso de prueba, el volcado de los ficheros de parse y el árbol.

Este es el enlace https://github.com/Dakixr/25-PDL/archive/main.zip para descargar el repositorio con la información requerida.

4. Diseño del Analizador Semántico

```
Axioma_ -> Axioma                {Axioma_.tipo := Axioma.tipo}
Axioma -> Sentencia Axioma       {if (Axioma.tipo = void) then Axioma.tipo :=
                                  Sentencia.tipo; else    Axioma.tipo := Sentencia.tipo x
                                  Axioma.tipo}
Axioma -> Funcion Axioma         {if (Axioma.tipo = void) then Axioma.tipo :=
                                  Funcion.tipo; else Axioma.tipo := Funcion.tipo x
                                  Axioma.tipo}
Axioma -> lambda                 {Axioma.tipo := void}
Sentencia -> S                   {Sentencia.tipo := S.tipo}
Sentencia -> IF_                 {Sentencia.tipo := IF_.tipo}
Sentencia -> let Tipo id ;       {insertarTipo(id.pos, Tipo.tipo); insertarDespl(id.pos,
                                  desp); desp := desp + Tipo.ancho; ZD = False}
IF_ -> IF1 IF2                   {if (IF1.tipo = tipo_ok AND (IF2.tipo = tipo_ok OR
                                  IF2.tipo = void)) then IF_.tipo := tipo_ok; else
                                  IF_.tipo := tipo_error}
IF1 -> if ( E ) Senten           {IF1.tipo := if(E.tipo = logico AND Senten.tipo =
                                  tipo_ok) then tipo_ok; else tipo_error}
IF2 -> else Senten               {IF2.tipo := Senten.tipo}
IF2 -> lambda                    {IF.tipo := void}
Senten -> Sentencia              {Senten.tipo := Sentencia.tipo}
Senten -> { Lista_Sentencias }   {Senten.tipo := Lista_Sentencias.tipo}
S -> id = E ;                    {if (obtenerTipo(id.pos) = E.tipo) then S.tipo :=
                                  tipo_ok; else S.tipo := tipo_error}
S -> id %= E ;                   {if (obtenerTipo(id.pos) = E.tipo AND E.tipo = ent) then
                                  S.tipo := tipo_ok; else S.tipo := tipo_error}
S -> id ( Parametros ) ;         {S.tipo := if (Parametros.tipo.length =
                                  buscaNumParam(id.pos) AND Parametros.tipo =
                                  buscarParamTipo(id.pos)) then tipo_ok; else tipo_error}
S -> alert ( E ) ;               {if (E.tipo = ent OR E.tipo = cadena) then S.tipo :=
                                  tipo_ok; else S.tipo := tipo_error}
S -> input ( id ) ;              {if (id.tipo = ent OR id.tipo = cadena) then S.tipo :=
                                  tipo_ok; else S.tipo := tipo_error}
S -> return X ;                  {if (X.tipo = obtenerTipoRet(id.pos)) then S.tipo :=
                                  tipo_ok; else S.tipo := tipo_error}
Parametros -> E K2               {if (K2.tipo = void) then Parametros.tipo := E.tipo ;
                                  else Parametros.tipo := E.tipo x K2.tipo}
Parametros -> lambda             {Parametros.tipo := void}
K2 -> , E K2                     {if (K2.tipo = void) then K2.tipo := E.tipo; else
                                  K2.tipo := E.tipo x K2.tipo}
K2 -> lambda                     {K2.tipo := void}
X -> E                           {X.tipo := E.tipo}
X -> lambda                      {X.tipo := void}
```

```
Funcion -> F1 F2 F3             {insertarTipoParams(id.pos, F2.tipo); F.tipo = F3.tipo;
                                 borrar_tabla_actual(); ZD =  False}
F1 -> function Tipo_B id        {insertarTipoRetorno(id.pos, Tipo_B.tipo); F1.tipo :=
                                 Tipo_B.tipo, crearTablaFun()}
F2 -> ( Cabecera )              {F2.tipo := Cabecera.tipo; ZD = False}
F3 -> { Lista_Sentencias }      {F3.tipo := Lista_Sentencias.tipo}
Tipo -> boolean                 {Tipo.tipo := logico, ZD = True}
Tipo -> number                  {Tipo.tipo := ent, ZD = True}
Tipo -> string                  {Tipo.tipo := cadena, ZD = True}
Tipo_B -> Tipo                  {Tipo_B.tipo := Tipo.tipo}
Tipo_B -> lambda                {Tipo_B.tipo := void}
Cabecera -> Tipo id K           {Cabecera.tipo := if (K.tipo = void) then Cabecera.tipo
                                 = Tipo.tipo; else Tipo.tipo x K.tipo;
                                 insertarTipo_desp(id,tipo.tipo,tipo.desp)}
Cabecera -> lambda              {Cabecera.tipo := void}
K -> , Tipo id K                {K.tipo := if (K.tipo = void) then K.tipo = Tipo.tipo;
                                 else K.tipo = Tipo.tipo x K.tipo,
                                 añadir_tipo_desp(id,tipo)}
K -> lambda                     {K.tipo := void}
Lista_Sentencias -> Sentencia Lista_Sentencias
                                {if (Sentencia.tipo and Lista_Sentencias.tipo !=
                                 tipo_error) Lista_Sentencias = tipo_ok else tipo_error}
Lista_Sentencias -> lambda      {Lista_Sentencias.tipo := void}
E -> ! V                        {if (V.tipo = logico) then E.tipo := logico; else E.tipo
                                 := tipo_error}
E -> R                          {E.tipo := R.tipo}
R -> R > U                      {if (R.tipo = U.tipo AND R.tipo = ent) then R.tipo :=
                                 logico; else R.tipo := tipo_error}
R -> R < U                      {if (R.tipo = U.tipo AND R.tipo = ent) then R.tipo :=
                                 logico; else R.tipo := tipo_error}
R -> U                          {R.tipo := U.tipo}
U -> U + V                      {if (U.tipo = V.tipo AND U.tipo = ent) then U.tipo :=
                                 ent; else U.tipo := tipo_error}
U -> U - V                      {if (U.tipo = V.tipo AND U.tipo = ent) then U.tipo :=
                                 ent; else U.tipo := tipo_error}
U -> V                          {U.tipo := V.tipo}
V -> id                         {V.tipo := obtenerTipo(id.pos)}
V -> ( E )                      {V.tipo := E.tipo}
V -> id ( Parametros )          {V.tipo := obtenerTipoRet(id.pos)}
V -> ent                        {V.tipo := ent}
V -> cadena                     {V.tipo := cadena}
```

5. Diseño de la Tabla de Símbolos

Para el diseño final de la Tabla de Símbolos, se ha seguido con lo ya indicado en el diseño inicial, realizado en el diseño del analizador semántico. Así, los parámetros de la tabla son los siguientes:

La estructura almacena toda la información relevante sobre los identificadores del programa:

- "index": indica la posición que ocupa en la tabla de símbolos
- "lexema": nombre del identificador
- "Tipo": indica el tipo del identificador (entero, array…)
- "Despl": dirección relativa
- "numParam": número de parámetros que tiene la función
- "TipoParamXX": tipos de parámetros que tiene la función. XX representa un número de dos dígitos, cuyos valores irán desde el 1 hasta el valor del atributo numParam.
- "TipoRetorno": tipo devuelto por la función
- "EtiqFuncion": etiqueta de comienzo del código de la función
- "Param": representa que un identificador local es un parámetro formal del subprograma donde está declarado.

Para los desplazamientos, se ha establecido que los enteros y los lógicos ocupan 1 palabra y las cadenas, 8.

Y siguiendo con lo anterior, se establecen los ámbitos de los identificadores. Así, cuando se crea una función, se creará una nueva tabla de símbolos para su propio ámbito y las variables locales que ahí se declaren, se introducirán en esa misma tabla. Cuando se vaya a usar una variable, se deberá buscar dentro del ámbito en el que se encuentra y también, en el caso de que se esté dentro de una función, en la tabla principal. Para crear una variable dentro de una función, solo se deberá comprobar si esa variable ya existe dentro de ese mismo ámbito, independientemente de si ya existe otra variable con el mismo identificador en la tabla principal, pues tendrán ámbitos distintos.

Una de nuestras mayores dificultades con la tabla de símbolos ha sido cuando se llama a una variable no inicializada y que, por tanto, ha de crearse como una variable global entera. Por tanto, en el caso de que se esté dentro del ámbito de una función, esta variable debe salirse y añadirse en la tabla de símbolos principal.

Para la manipulación de las estructuras internas de la tabla de símbolos, se han desarrollado diferentes funciones que usarán el analizador léxico, para añadir lexemas a la tabla; y el analizador semántico, para añadir otra información relevante de los lexemas (tipo, desplazamiento, parámetros, etc.).

```
def restore_state():
    '''Devolver la tabla de simbolos a su estado inicial.'''
def get_index(lex, var_local = False):
    '''Devolver el index del identificador'''
def get_lex(index):
    '''Devolver el lexema del index'''
def get_return_type(index):
    '''Devolver el tipo de retorno de la función'''
def get_tipo(index):
    '''Devolver el tipo del identificador'''
def get_curr_function():
    '''Devolver el id de la función ejecutandose.'''
def get_list_num_params(index):
    '''Devuelve una lista con los parametros de la función y el
       número de param.'''
def add_lex(lex, var_global = False):
    '''Devolver el index del identificador y si no existe,
       crear una entrada en la lista_tabla de símbolos.'''
def crear_tabla(name_funcion):
    '''Crear una nueva tabla de simbolos'''
def borrar_current_tabla():
    '''Eliminar del stack la tabla correspondiente.'''
def add_tipo_desplazamiento(index, tipo):
    '''Añadir un tipo y desplazamiento a la tabla de simbolos de
       un determinado ID.'''
def add_tipo_num_parametros(index, tipo_param_str):
    '''Añade los tipos de parametros a la tabla se símbolos.'''
def add_return_type_and_type(index, tipo_retorno):
    '''Añadir un tipo de retorno a la tabla de simbolos del id s
       eleccionado y tipo función.'''
def save_symbol_table():
    '''Guardar a un archivo externo "symbol_table.txt" la tabla
       de simbolos.'''
```

6. Anexo

La implementación del código aquí explicado se encuentra adjunta junto a este mismo documento. También puede encontrarse en el enlace siguiente: [enlace de GitHub]

Casos de prueba correctos

Se han publicado en GitHub cinco casos de prueba correctos de nuestro procesador de lenguajes y que se encuentra en el siguiente enlace:

- Casos correctos.

Caso 1

En este caso, se prueba el funcionamiento de la precedencia de las operaciones y se observa un funcionamiento correcto.

Caso 2

Ahora, se prueba un programa simple con variables, alert, input y funciones.

Caso 3

En este caso, se vuelve a probar un programa simple con variables, alert, input y funciones.

Caso 4

Aquí, además de seguir comprobando lo ya indicado en los ejemplos anteriores, se añaden operaciones con módulo (%=) y negación (!) y sentencias if.

Caso 5

En este último caso correcto, también probamos la sentencia if-else y la diferenciación entre mayúsculas y minúsculas (lenguaje sensitive) de JavaScriptPDL.

Casos de prueba erróneos

Se han desarrollado cinco casos de prueba de errores que son detectados y notificados correctamente y que se encuentran en el siguiente enlace:

- Casos con errores.

Caso 1

Para este caso, se producen cuatro errores semánticos, que el analizador detecta y notifica correctamente.

Caso 2

En este caso, se producen erróneamente asignaciones de tipos incorrectos y negaciones de operaciones no booleanas, que detectará el analizador semántico.

Caso 3

Ahora, el problema radica en que se escriben dos puntos y coma para señalizar el fin de una sentencia, que detecta y notifica el analizador léxico.

Caso 4

En este caso, se detecta que se declara una variable que ya estaba previamente declarada y asignada.

Caso 5

En este último caso, se instancia una función sin las correspondientes comas entre los argumentos de esta, de forma que el analizador sintáctico lo detecta y notifica.