Procesadores de Lenguajes

PL-51

Componentes del grupo PL-51:
Lucía Mondéjar Morcillo, 120079
Borja Peiró Villanueva, 110387
Curso 2016/2017

1. Índice

2. Descripción del diseño del Procesador

Para la práctica de la asignatura Procesadores de Lenguajes, del quinto semestre, deberemos diseñar e implementar un procesador para el lenguaje JavaScript-PL. Nuestro grupo, el grupo PL-51 tiene asignadas: la sentencia condicional compuesta if, if-else, el operador especial asignación con y lógico (&=) y la técnica de Análisis Sintáctico Descendente Recursivo.

A la hora de llevarla a cabo, hemos elegido Python para generar el código y la herramienta VASt, que se utilizará para visualizar los árboles sintácticos que parten de los ficheros de la gramática y el parse.

En esta memoria se recogen todos los analizadores de los que componen un compilador. Este compilador tiene un Analizador Léxico, un Analizador Sintáctico, un Analizador Semántico y una Tabla de Símbolos.

El contenido del lenguaje, se divide en varias partes:

- **Comentarios**: Tenemos dos tipos de comentarios, /* … */ y // … . En nuestro caso, hemos implementado ambos.

- **Constantes**: Enteros, cadenas de caracteres y lógicas. En el caso de los enteros, son los números enteros naturales comprendidos entre 0 y 32767. Las cadenas de caracteres vienen delimitadas por "…". Para las constantes lógicas, se han implementado las palabras reservadas True y False.

- **Operadores**: Aritméticos, de relación, lógicos, y de asignación. En nuestro caso, hemos implementado la suma (A + B) para el operador aritmético, mayor (A > B) para el operador de relación, operación AND (A && B) para el operador lógico, operación AND con asignación (A &= B), como operador asignado; y la asignación (=) como operador de asignación.

- **Identificadores**: Los identificadores están formados por cualquier cantidad de letras, dígitos, siendo siempre el primero una letra.

- **Declaraciones**: El lenguaje utilizado, JavaScript-PL no exige que todas las variables estén declaradas. Si se usa una variable que no ha sido declarada anteriormente, se considera que esa variable es global y entera. Si se declara una variable, tendrá que ser de la forma "var T id", siendo T el tipo de variable (entera, lógica o cadena).

- **Tipos de Datos**: Para la realización de la práctica ha considerado sólo la existencia de tres tipos de datos: entero (int), cadena (chars) y lógico (bool). El lenguaje no tiene conversiones automáticas entre tipos, es decir, no podemos hacer castings.

- **Instrucciones de Entrada/Salida**: La sentencia write (expresión) evalúa la expresión e imprime el resultado por la pantalla. La sentencia prompt (var) lee un número o una cadena del teclado y lo almacena en la variable var, que tiene que ser de tipo entero o cadena.

- **Sentencias**: Hay distintos tipos de sentencias en nuestra práctica, sentencia de asignación, sentencia de llamada a una función, sentencia de retorno de una función, sentencia condicional compuesta y simple, sentencia de selección múltiple, sentencia de asignación, etc. A nuestro grupo nos fue asignado la sentencia condicional compuesta, además de la de asignación con condición lógica (&=).

- **Funciones**: Se define indicando la palabra “function”, el tipo de retorno (en el caso de que devuelve algo), el nombre de la función y los argumentos con sus tipos entre paréntesis en el caso que haya argumentos. El cuerpo de la función va delimitado entre llaves.

3. Diseño del Analizador Léxico

El Analizador Léxico se encarga de manejar el fichero fuente. Cuando encuentra un token, lo manda al Analizador Sintáctico. Además, se encarga de tratar los errores léxicos, y situarlos en el fichero fuente. Obvia los blancos, otros delimitadores y los comentarios.

Dentro de este Analizador, encontraremos distintas partes: los tokens, la gramática, el autómata generado por la gramática, las acciones semánticas y los errores.

3.1. Tokens

Los tokens son, como hemos comentado anteriormente, los elementos más pequeños del texto.
El formato de los tokens será **del* < código del* , del* [atributo] del* > del* RE**, donde del* será cualquier cantidad de espacios en blanco, tabuladores o vacío. El código indica el código del token correspondiente, con el siguientes formato: $(l|d)^+$, que son los caracteres alfanuméricos, habiendo al menos uno. El atributo es el atributo opcional del token que corresponde, que puede tener un formato cualquiera de los siguientes: un nombre, $(l|d)^+$ caracteres alfanuméricos, habiendo al menos uno; un número: $[+|-]d^+$ que implica un número entero con signo opcional, o una cadena "c^+" que es una cadena de caracteres no vacía. RE puede implicar un salto de línea (CR - Carriage Return) o un fin de fichero (EOF - End Of File).

Los tokens que reconoce nuestro analizador léxico son:
< IniPar, (>: Inicio de paréntesis
< FinPar,) >: Fin de paréntesis
< IniLla, { >: Inicio de llaves
< FinLla, } >: Fin de llaves
< cr, - > : Salto de línea
< id, lexema >: Identificador
< PalRes, bool >: Palabra reservada bool
< PalRes, chars >: Palabra reservada chars
< PalRes, function >: Palabra reservada function
< PalRes, if > : Palabra reservada if
< PalRes, else > : Palabra reservada else
< PalRes, int >: Palabra reservada int
< PalRes, prompt >: Palabra reservada prompt
< PalRes, return >: Palabra reservada return
< PalRes, write >: Palabra reservada write
< PalRes, True >: Palabra reservada true
< PalRes, False >: Palabra reservada false
< PalRes, var >: Palabra reservada var
< Coma, “,” >: Coma
< OpAsi, “=” >: Operador asignación

< OpLogico, “&&” >: Operador lógico AND
< OpEspcial, “&=” >: Operador especial asignación con y lógico
< OpNum, “>” >: Operador relacional MAYOR
< Sum, + >: Suma
< FinFich, eof >: Fin de fichero
< entero, valor >: Número con un valor
< cadena, texto >: Cadena con un texto

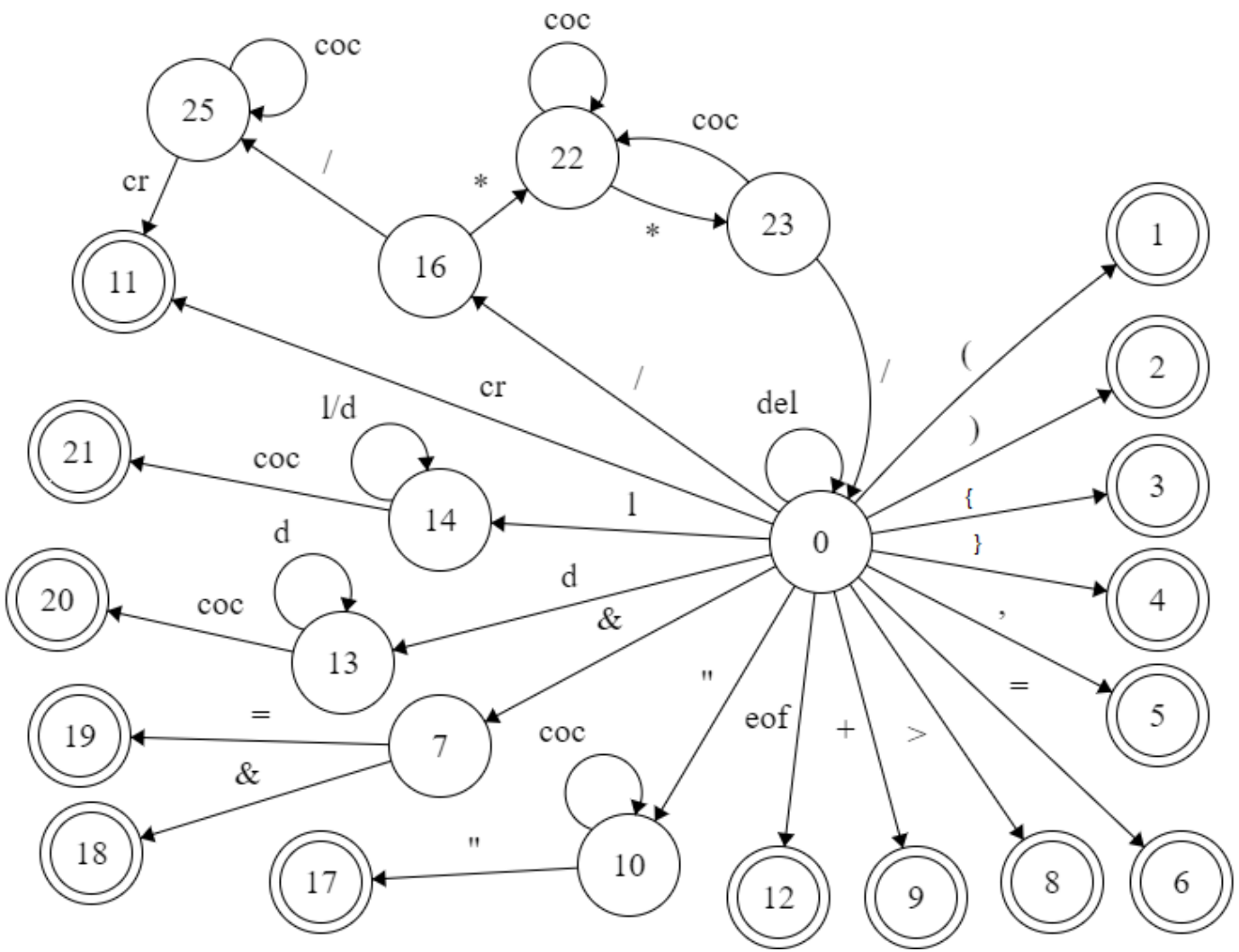
3.2. Gramática

A continuación, encontramos la gramática:
coc: Cualquier otro carácter.
l: Letra.
d:Dígito.

S → del S | (|) | { | } | , | = | &A | > | + | “B | cr | eof | dC | lD | /E
A → & | =
B → cocB | “
C → dC | λ
D → lD | dD | λ
E → *F | /I
F → coc F | *H
H → coc F | /
I → coc I | cr

3.3. Autómata

El autómata que reconoce la gramática es el siguiente:



3.4. Acciones semánticas

Las acciones semánticas de nuestro Analizador Léxico son las siguientes:

0:0 – leer
0:1 – leer, generarToken(IniPar,')')
0:2 – leer, generarToken(FinPar,'(')
0:3 – leer, generarToken(IniLla,'}')
0:4 – leer, generarToken(FinLla,'{')
0:5 – leer, generarToken(coma, ',')
0:6 – leer, generarToken(OpAsi, '=')
0:8 – leer, generarToken(OpNum, '>')
0:9 – leer, generarToken(Sum,'+')
0:12 – leer, generarToken(eof, -)
0:10 – leer
10:10 – leer, concatena(lex, coc.valor)
10:17 – leer, generarToken(cadena, lex)
0:7 – leer
7:18 – leer, generarToken(OpLogico, '&&')
7:19 – leer, generarToken(OpEspcial, '&=')
0:13 – leer, valor = d
13:13 – leer, valor = valor*10 + d
13:20 – if (valor > 32768)
Then Error ("valor demasiado alto")
else
Then generarToken(entero, valor)
0:14 – leer, concatena(lex, l)
14:14 – leer, x=l/d, concatena(lex, x)
14:21 – if (buscarPalRes(lex) == True)
Then generarToken(PalRes, lex)
else
Then generarToken(id, lex)
0:11 – leer, generarToken(cr, -)
0:16 – leer
16:25 – leer
25:25 – leer
25:11 – leer, generarToken(cr,-)
16:22 – leer
22:22 – leer
22:23 – leer
23:22 – leer
23:0 – leer

3.5. Errores

En nuestro Analizador Léxico hemos tratado dos tipos de errores, el primero de ellos es el número máximo que un valor de tipo entero puede tener, que es 32767 y el segundo de ellos es que cualquier otro carácter que no esté contemplado en nuestras acciones semánticas es reconocido como id, que posteriormente dará error en el analizador sintáctico.

4. Diseño del Analizador Sintáctico

En este apartado, demostraremos que la gramática es adecuada para el Analizador Sintáctico Descendente Recursivo Predictivo.

4.1. Gramática

Para que la gramática sea válida para nuestro Analizador Sintáctico Descendente Recursivo Predictivo, deberemos comprobar que no es recursiva por la izquierda, y deberá estar factorizada. Una vez que cumpla dichos requisitos, procederemos a comprobar los conflictos.

La gramática en cuestión es la siguiente:

P' → P
P → B Z P | F Z P | Z P | eof
Z → cr
F → function H id (A) Z { Z C }
H → T | **λ**
T → int | bool | chars
A → T id K | **λ**
K → , T id K | **λ**
B → var T id | if (E) { Z C } B' | S // sentencias
B'→ else { Z C } | **λ**
S → id S'| return X | write (E) | prompt (id)
S'→ (L) | = E
X → E | **λ**
C → B Z C | **λ** // cuerpo de la función
L → E Q | **λ**
Q → , E Q | **λ**
E → R E'
E'→ && R E' | &= R E' | **λ**
R → U R'
R'→ > U R'| **λ**
U → V U'
U'→ + V U' | **λ**
V → id V' | (E) | entero | cadena | True | False
V'→ (L) | **λ**

Para la resolución de nuestro ejercicio hemos diseñado la siguiente gramática que contempla todos los casos posibles. Hemos encontrado recursividad por la izquierda en los siguientes casos:
E → E && R | E &= R | R
R → R > U | U
U → U + V | V

Al eliminar la recursividad por la izquierda, quedaría de la siguiente forma:

E → R E'
E'→ && R E' | &= R E' | **λ**
R → U R'
R'→ > U R'| **λ**
U → V U'
U'→ + V U' | **λ**

4.2. First

A continuación, indicamos los Firsts de la gramática:

First (P') = {var, if, return, write, prompt, id, function, cr, eof}
First (P) = {var, if, return, write, prompt, id, function, cr, eof}
First (Z) = {cr}
First (F) = {function}
First (H) = {int, bool, chars, λ}
First (T) = {int, bool, chars}
First (A) = {int, bool, chars, λ}
First (K) = {",", λ}
First (B) = {var, if, return, write, prompt, id}
First (B') = {else, λ}
First (S) = {return, write, prompt, id}
First (S') = {(, =}
First (X) = {(, id, entero, cadena, True, False, λ}
First (C) = { var, if, return, write, prompt, id, λ}
First (L) = {(, id, entero, cadena, True, False, λ}
First (Q) = {",", λ}
First (E) = {(, id, entero, cadena, True, False}
First (E') = {&&, &=, λ}
First (R) = {(, id, entero, cadena, True, False}
First (R') = {>, λ}
First (U) = {(, id, entero, cadena, True, False}
First (U') = {+, λ}
First (V) = {(, id, entero, cadena, True, False}
First (V') = {(, λ}

4.3. Follow

A continuación, indicamos los Follow de la gramática:

Follow (P') = {$}
Follow (P) = {$}
Follow (Z) = {var, if, id, return, write, prompt, function, cr, eof, "}", else }
Follow (F) = {cr}
Follow (H) = {id}
Follow (T) = {id}
Follow (A) = {")"}
Follow (K) = {")"}
Follow (B) = {cr}
Follow (B') = {cr}
Follow (S) = {cr}
Follow (S') = {cr}
Follow (X) = {cr}
Follow (C) = {cr}
Follow (L) = {cr}
Follow (Q) = {cr}
Follow (E) = {")", cr, ","}
Follow (E') = {")", cr, ","}
Follow (R) = {&&, &=, ")", cr, ","}
Follow (R') = {&&, &=, ")", cr, ","}
Follow (U) = {>, &&, &=, ")", cr, ","}
Follow (U') = {>, &&, &=, ")", cr, ","}
Follow (V) = {+, >, &&, &=, ")", cr, ","}
Follow (V') = {+, >, &&, &=, ")", cr, ","}

4.4. Conflictos

Estudiaremos si existen conflictos en la gramática descrita en al inicio de este apartado:

P → First (B) ∩ First (F) ∩ First (Z) ∩ eof =
= {var, if, return, write, prompt, id} ∩ {function} ∩ {cr} ∩ eof = ø

H → First (T) ∩ Follow (H) = {int, bool, chars} ∩ {id} = ø

A → First (T) ∩ Follow (A) = {int, bool, chars} ∩ {“)”} = ø

K → “,” ∩ Follow (K) = “,” ∩ {“)”} = ø

B → var ∩ if ∩ First (S) = var ∩ if ∩ {return, write, prompt, id} = ø

B’ → else ∩ Follow (B’) = else ∩ {cr} = ø

S → id ∩ return ∩ write ∩ prompt = ø

S’ → “(“ ∩ “=“ = ø

T → int ∩ chars ∩ bool = ø

X → First (E) ∩ Follow (X) = {(, id, entero, cadena, True, False} ∩ {cr} = ø

C → First (B) ∩ Follow (C) = {var, if, return, write, prompt, id} ∩ {cr} = ø

L → First (E) ∩ Follow (L) = {(, id, entero, cadena, True, False} ∩ {cr} = ø

Q → “,” ∩ Follow (Q) = “,” ∩ {cr} = ø

E’→ && ∩ &= ∩ Follow (E’) = && ∩ &= ∩ {“)”, cr, “,”} = ø

R’→ > ∩ Follow (R’) = > ∩ {&&, &=, “)”, cr, “,”} = ø

U’→ + ∩ Follow (U’) = + ∩ {>, &&, &=, “)”, cr, “,”} = ø

V→ id ∩ “(“ ∩ entero ∩ cadena ∩ True ∩ False = ø

V’→ “(“ ∩ Follow (V’) = “(“ ∩ {+, >, &&, &=, “)”, cr, “,”} = ø

4.5. Procedures

```
PROCEDURE ComprobarToken(token):
  BEGIN
    If token = sigueiten_token
         Then
              Leer_siguiente_token(siguiente_token)
    Else
         Then
              Error_Sintactico()
  END

PROCEDURE P':
  BEGIN
    P;
  END

PROCEDURE P:
  BEGIN
    B;
    Z;
    P;

  END

PROCEDURE Z:
  BEGIN
    ComprobarToken(cr)
  END

PROCEDURE F:
  BEGIN
    ComprobarToken(function )
    H()
    ComprobarToken(id)
    ComprobarToken("(")
    A()
    ComprobarToken(")")
    Z()
    ComprobarToken("{")
    Z()
    C()
    ComprobarToken("}")
  END

PROCEDURE H:
  BEGIN
    if siguiente_token = ( "int" or "bool" or "chars" ) :
         Then
         T()
  END
```

```
PROCEDURE T:
  BEGIN
    if siguiente_token = "int" :
          Then
                ComprobarToken("int")

    else if siguiente_token = "bool" :
          Then
                ComprobarToken("bool")
    else :
          Then
                ComprobarToken("chars")

  END

PROCEDURE A:
  BEGIN
    if siguiente_token = ( "int" or "bool" or "chars" ) :
          Then
          T()
          ComprobarToken("id")
          K()
  END

PROCEDURE K:
  BEGIN
    if siguiente_token = "," :
          Then
                ComprobarToken(",")
                T()
                ComprobarToken("id")
                K()
  END

PROCEDURE B:
  BEGIN
    if siguiente_token = "var" :
          Then
                ComprobarToken("var")
                T()
                ComprobarToken("id")
    else if siguiente_token = "if" :
          Then
                ComprobarToken("if")
                E()
                ComprobarToken("{")
                Z()
                C()
                ComprobarToken("}")
                B'()
    else
                S()
  END
```

```
PROCEDURE B':
  BEGIN
    if siguiente_token = "else" :
          Then
                ComprobarToken("else")
                ComprobarToken("{")
                Z()
                C()
                ComprobarToken("}")
  END

PROCEDURE S:
  BEGIN
    if siguiente_token = "id" :
          Then
                ComprobarToken("id")
                S'()
    else if siguiente_token = "return" :
          Then
                ComprobarToken("return")
                X()
    else if siguiente_token = "write" :
          Then
                ComprobarToken("write")
                ComprobarToken(")")
                E()
                ComprobarToken(")")

    else
                ComprobarToken("prompt")
                ComprobarToken("(")
                ComprobarToken("id")
                ComprobarToken(")")

  END

PROCEDURE S':
  BEGIN
    if siguiente_token = "(" :
          Then
                ComprobarToken("(")
                L()
                ComprobarToken(")")
    else :
          Then
                ComprobarToken("=")
                E()
  END

PROCEDURE X:
  BEGIN
    if siguiente_token = ( "(" or "id" or "entero" or "cadena"
or "True" or "False") :
          Then
                E()
  END
```

```
PROCEDURE C:
  BEGIN
    if siguiente_token = (“var” or “if” or “return” or “write”
or “prompt” or “id”) :
          Then
                B()
                Z()
                C()
  END

PROCEDURE L:
  BEGIN
    if siguiente_token = ( “(“ or “id” or “entero” or “cadena”
or “True” or “False”) :
          Then
                E()
                Q()
  END

PROCEDURE Q:
  BEGIN
    if siguiente_token = “,” :
          Then
                ComprobarToken(“,”)
                E()
                Q()
  END

PROCEDURE E:
  BEGIN
    if siguiente_token = ( “(“ or “id” or “entero” or “cadena”
or “True” or “False”) :
          Then
                R()
                E’()
  END

PROCEDURE E’:
  BEGIN
    if siguiente_token = “&&” :
          Then
                R()
                E’()
    if siguiente_token = “&=” :
          Then
                R()
                E’()
  END

PROCEDURE R:
  BEGIN
          U()
          R’()
  END
```

```
PROCEDURE R':
  BEGIN
    if siguiente_token = ">" :
          Then
                ComprobarToken(">")
                U()
                R'()
  END

PROCEDURE U:
  BEGIN
          V()
          U'()
  END

PROCEDURE U':
  BEGIN
    if siguiente_token = "+" :
          Then
                ComprobarToken("+")
                V()
                U'()
  END

PROCEDURE V:
  BEGIN
    if siguiente_toke = "id" :
          Then
                ComprobarToken("id")
                V'()
    else if siguiente_token = "(" :
          Then
                ComprobarToken("(")
                E()
                ComprobarToken(")")
    else if siguiente_token = "entero" :
          Then
                ComprobarToken("entero")
    else if siguiente_token = "cadena" :
          Then
                ComprobarToken("cadena")
    else if siguiente_token = "True" :
          Then
                ComprobarToken("True")
    else :
          Then ComprobarToken("False")
  END

PROCEDURE V':
  BEGIN
    if siguiente_token = "(" :
          Then
                ComprobarToken("(")
                L()
                ComprobarToken(")")
  END
```

5. Diseño del Analizador Semántico

El Analizador semántico se encarga de estudiar el significado de todos los elementos dentro del texto, además de efectuar las comprobaciones sensibles al contexto gracias a la Tabla de Símbolos.
Se usan las siguientes funciones:

buscarTS => busca si existe TS en DicTS, si no devuelve null
buscarId => busca si exixte el id en la TS actual, en caso contrario devuelve null
buscarTipoId => busca si existe el id en la TS actual y devuelve su tipo, si no existe devuelve null
buscarTipoFuncion => busca si existe el id en la TS actual y es de tipo "function" y devuelve la salida de la funcion, si no existe, o no es funcion, devuelve null
buscarTipoParametrosFuncion => busca si existe el id en la TS actual y es de tipo "function" y devuelve el tipo de parametros que se le pasa, si no existe, o no es funcion, devuelve null
insertarTS => inserta un id en la TS actual
insertarFunTs => inserta los parametros de una funcion dado un ptrTS valido

P' →
{{
ptrTS_actual = "TS_General"
DicTS[ptrTS_actual] = [[], 0, "null"] // (TS,Desp,TS_padre)
}}
P
{{
TS_actual = "null"
}}

P → B Z P (No hace nada)
P → F Z P (No hace nada)
P → Z P (No hace nada)
P → eof (No hace nada)
Z → cr (No hace nada)

F → function H id
{{
if buscarTS(id.ent) ≠ null
Then Error (“Función ya declarada”)
else
insertarTS(id.ent, "function", Desp_actual, ptrTS_actual)
DicTS[ptrTS_actual] := (TS_actual,Desp_actual) // guarda la TS actual con el desplazamiento
crearTS TS_nueva
TS_actual := TS_nueva
Desp_actual := 0
ptrTS_anterior := ptrTS_actual
ptrTS_actual := id.ent
}}

```
        (A) Z {
        {{
        insertarFunTS(ptrTS_anterio, id.ent, A.numParam, A.tipo, id.ent)
        }}
        Z C }
        {{
        if C.tipo ≠ H.tipo or C.tipo ≠ tipo_ok
                Then Error ("Funcion mal return")
        else
                Then
                DicTS[ptrTS_actual] := (TS_actual,Desp_actual) // guarda la TS actual
con el desplazamiento
                DicTS[ptrTS_anterior] := (TS_anterior,Desp_anterior) // recuperamos la
tabla de simbolos anterior
                TS_actual       := TS_anterior
                Desp_actual := Desp_anterior
                ptrTS_actual  := ptrTS_anterio
                ptrTS_anterio := null
        }}

H → T                   {{ H.tipo = T.tipo , H.tamano = T.tamano }}
H → λ       {{ H.tipo = tipo_vacio  , H.tamano = 0 }}

T → int         {{ T.tipo = entero , T.tamano = 2 }}
T → bool        {{ T.tipo = logico , T.tamano = 1 }}
T → chars       {{ T.tipo = cadena  , T.tamano = 2 }}

A → T id
        {{
        if buscarId(id.ent) ≠ null
                Then Error ("Identificador ya creado")
        else
                Then
                        insertarTS(id.ent, T.tipo, Desp_actual, ptrTS_actual)
                        Desp_actual += T.tamano
        }}

        K

        {{
        if K.tipo == "tipo_vacio"
                Then A.tipo := T.tipo
        else
                Then
                        A.tipo := f(T.tipo, K.tipo)
                        A.numParam := K.numParam + 1
        }}

A → λ
        {{
```

```
        A.tipo = tipo_vacio
        A.numParam := 0
        }}

K0 → , T id
        {{
        if buscarId(id.ent) ≠ null
                Then Error (“Identificador ya creado”)
        else
                Then
                        insertarTS(id.ent, T.tipo, Desp_actual, ptrTS_actual)
                        Desp_actual += T.tamano
        }}

        K1

        {{
        if K1.tipo == “tipo_vacio”
                Then K0.tipo := T.tipo
        else
                Then
                        K0.tipo := f(T.tipo, K1.tipo)
                        K0.numParam := K1.numParam + 1
        }}

K → λ
        {{
        K.tipo = tipo_vacio
        K.numParam := 0
        }}

B → var T id
        {{
        if buscarId(id.ent) ≠ null
                Then Error (“Identificador ya creado”)
        else
                Then
                        insertarTS(id.ent, T.tipo, Desp_actual, ptrTS_actual)
                        Desp_actual += T.tamano
                        B.tipo := tipo_ok
        }}

B → if (E)
        {{
        if E.tipo == logico
                Then B.tipo := tipo_ok
        else
                Then Error (“condición "if" no logica”)
        }}
```

```
        { Z C } B'

        {{
        if C.tipo ≠ tipo_ok and C.tipo ≠ tipo_vacio
                Then Error (“contenido "if" mal formado”)
        }}

B → S
        {{
        B.tipo := S.tipo
        }}

B'→ else { Z C }
        {{
        if C.tipo ≠ tipo_ok
                Then Error (“contenido "else" mal formado”)
        else
                Then B'.tipo := tipo_ok
        }}

B'→ λ
        {{
        B'.tipo := tipo_vacio
        }}

S → id S'
        {{
        if buscarTipoId(id.ent) ≠ null
          Then
            if  buscarTipoFuncion(id.ent, ptrTS_actual) ≠ null
                Then
                  if buscarTipoParametrosFuncion(id.ent, ptrTS_actual) == S’.tipo:
                        Then S.tipo := buscarTipoFuncion(id.ent, ptrTS_actual)
                  else:
                        Then Error_Sem()
            else:
                Then
                  if buscarTipoId(id.ent, ptrTS_actual) == S’.tipo :
                        Then    S.tipo := tipo_ok
                  else:
                        Then    Error_Sem()
        else:
          Then
            insertarTS( id.ent, "entero" , DicTS["TS_General"][1], "TS_General")
            DicTS["TS_General"][1] += 2
            if "entero" == S’.tipo :
                S.tipo:= tipo_ok
            else:
                Error_Sem()
        }}
```

S → return X
{{
S.tipo := X.tipo
}}

S → write (E)
{{
S.tipo := E.tipo
}}

S → prompt (id)
{{
if buscarTipoId(id.ent) ≠ "entero" or buscarTipoId(id.ent) ≠ "cadena"
Then Error (“uso incorrecto de "prompt" ”)
else
Then S.tipo:= buscarTipoId(id.ent)
}}

S'→ (L)
{{
S'.tipo:= L.tipo
}}

S'→ = E
{{
S'.tipo := E.tipo
}}

X → E
{{
X.tipo := E.tipo
}}

X → **λ**
{{
X.tipo := tipo_vacio
}}

C0 → B Z C1
{{
if C1.tipo ≠ tipo_error and B.tipo ≠ tipo_error
Then C0.tipo := tipo_ok
else
Then Error (“contenido de corchetes mal formado”)
}}

C → **λ**
{{
C.tipo := tipo_vacio

```
        }}

L → E Q
        {{
        if E.tipo ≠ tipo_error and Q.tipo ≠ tipo_error
                Then L.tipo:= f(E.tipo,Q.tipo)
        else
                Then Error (“contenido parentesis del identificador llamado mal
formado”)
        }}

L → λ
        {{
        L.tipo := tipo_vacio
        }}

Q0 → , E Q1
        {{
        if E.tipo ≠ tipo_error and Q1.tipo ≠ tipo_error
                Then Q0.tipo:= f(E.tipo,Q.tipo)
        else
                Then Error (“contenido parentesis del identificador llamado mal
formado”)

        }}

Q → λ
        {{
        Q.tipo := tipo_vacio
        }}

E → R E'
        {{
        if E'.tipo ≠ tipo_vacio and R.tipo ≠ E'.tipo
                Then Error (“Comparacion de elementos con tipos distintos”)
        else
                Then
                        if E'.tipo ≠ tipo_vacio
                                Then E.tipo := "logico"
                        else
                                Then E.tipo := R.tipo

        }}

E'0→ && R E'1
        {{
        if E'1.tipo ≠ tipo_vacio and E'1.tipo ≠ R.tipo ≠ "logico"
                Then Error (“Comparacion de elementos de tipo no logico”)
        else
                Then E'0.tipo := R.tipo
```

```
        }}

E'0→ &= R E'1

        {{
        if E'1.tipo ≠ tipo_vacio and E'1.tipo ≠ R.tipo ≠ "logico"
                Then Error (“Comparacion de elementos con asignacion de tipo no
logico”)
        else
                Then E'0.tipo := R.tipo
        }}

E'→ λ
        {{
        E'.tipo := tipo_vacio
        }}

R → U R'
        {{
        if R'.tipo ≠ tipo_vacio and U.tipo ≠ R'.tipo
                Then Error (“Comparacion de elementos con tipos distintos”)
        else
                Then
                        if R'.tipo ≠ tipo_vacio
                                Then R.tipo := "logico"
                        else
                                Then R.tipo := U.tipo

        }}

R'0→ > U R'1
        {{
        if R'1.tipo ≠ tipo_vacio and R'1.tipo ≠ U.tipo ≠ "entero"
                Then Error (“Comparacion de elementos de tipo no entero”)
        else
                Then R'0.tipo := U.tipo
        }}

R'→ λ
        {{
        R'.tipo := tipo_vacio
        }}

U → V U'
        {{
        if U'.tipo ≠ tipo_vacio and V.tipo ≠ U'.tipo
                Then Error (“Suma de elementos de tipo no entero”)
        else
                Then U.tipo := V.tipo
        }}
```

```
U'0→ + V U'1
        {{
        if V.tipo ≠ "entero"
                Then Error (“Suma de elementos de tipo no entero”)
        else
                Then U'0.tipo := V.tipo
        }}

U'→ λ
        {{
        U'.tipo := tipo_vacio
        }}

V → id V'
        {{
        if buscarTipoId(id.ent, ptrTS_actual) ≠ null
          Then
            if  buscarTipoFuncion(id.ent, ptrTS_actual) ≠ null
              Then
                if uscarTipoParametrosFuncion (id.ent, ptrTS_actual) == V'.tipo
                      Then V.tipo := buscarTipoFuncion(id.ent, ptrTS_actual)
                else
                      Then Error (“Parametros mal insertados”)
            else
              Then V.tipo := buscarTipoId(id.ent)
        else
           Then
                insertarTS( id.ent, "entero" , DicTS["TS_General"][1], "TS_General")
                DicTS["TS_General"][1] += 2
                V.tipo := entero
        }}

V → (E)        {{ V.tipo := E.tipo }}
V → entero     {{ V.tipo := entero }}
V → cadena     {{ V.tipo := cadena }}
V → True       {{ V.tipo := logico }}
V → False      {{ V.tipo := logico }}

V'→ (L)
        {{
        V'.tipo := L.tipo
        }}

V'→ λ
        {{
        V'.tipo := tipo_vacio
        }}
```

6. Diseño de la Tabla de Símbolos

La Tabla de Símbolos (TS) es una colección de símbolos que utiliza nuestro texto para almacenar información de cada token. El procesador del lenguaje interactúa con ella para almacenar y consultar su contenido.

El lenguaje que describe las TS no distingue minúsculas y mayúsculas, salvo en las cadenas. El lenguaje tiene dos palabras reservadas: “LEXEMA” y “ATRIBUTOS”.

Un fichero con la TS podrá tener líneas en blanco o líneas con la información de la TS. Cada línea de información debe acabar obligatoriamente con un salto de línea. Existen tres tipos de líneas obligatorias, cada una de ellas con un formato diferente:

Línea con el número de la TS:
Esta línea se utiliza como encabezado para comenzar cada TS. El formato de esta línea es: **pal* # del* núm del* : del* RC**

Línea del lexema: Esta línea se utiliza para indicar cada entrada de la TS. El formato de esta línea es: **del* * del* [LEXEMA del* :] del* 'nombre' del* RC**

Esta línea se utiliza para indicar cada atributo perteneciente a la entrada previa indicada en la ‘línea del lexema’ de la TS. El formato de esta línea es: **del* + del* atributo del* : del* valor RC**

Ejemplo valido:
CONTENIDO DE LA TABLA # 1 :
* LEXEMA : 'apellidos2'
  ATRIBUTOS :
  + tipo : (esto es de tipo cadena) 'string'
  + desplazamiento : 16
  --------- ----------
* LEXEMA : 'valido#'
  ATRIBUTOS :
  + tipo : 'bool' (lógico)
  + desplazamiento : 48
  --------- ----------
 * 'nombre'
  + tipo : 'string'
  + desplazamiento : 0

7. Casos de Prueba

Caso 1

var int a

- Tabla de símbolos

CONTENIDO DE LA TABLA # TS_General :

```
* LEXEMA : a
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
================================================
```

- Tokens

```
< PalRes , var >
< PalRes , int >
< id , a >
< cr , - >
< FinFich , eof >
```

- Análisis parse

Des 1 2 17 10 6 5

Caso 2

var int a
a = 5
var int b
b = 6

- Tabla de símbolos

CONTENIDO DE LA TABLA # TS_General :

```
* LEXEMA : a
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
* LEXEMA : b
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 2
  + Tam_parametros : null
```

+ Tipo_parametro : null
+ Retorno_funcion : null
+ ptrTS_padre : null
--------- ----------
==========

- Tokens

< PalRes , var >
< PalRes , int >
< id , a >
< cr , - >
< id , a >
< OpAsi , = >
< entero , 5 >
< cr , - >
< PalRes , var >
< PalRes , int >
< id , b >
< cr , - >
< id , b >
< OpAsi , = >
< entero , 6 >
< cr , - >
< cr , - >
< FinFich , eof >

- Análisis parse

Des 1 2 17 10 6 2 19 22 27 36 40 43 48 45 42 39 6 2 17 10 6 2 19 22 27 36 40 43 48 45 42 39 6 4 6 5

Caso 3

```
var bool n
n = True

if (n &= True) {
}
```

- Tabla de símbolos
  CONTENIDO DE LA TABLA # TS_General :

  * LEXEMA : n
    ATRIBUTOS :
    + tipo : logico
    + desplazamiento : 0
    + Tam_parametros : null
    + Tipo_parametro : null
    + Retorno_funcion : null
    + ptrTS_padre : null
    --------- ----------
    ==========

- Tokens

< PalRes , var >
< PalRes , bool >
< id , n >
< cr , - >
< id , n >
< OpAsi , = >
< PalRes , True >
< cr , - >
< cr , - >
< PalRes , if >
< IniPar , (>
< id , n >
< OpEspcial , &= >
< PalRes , True >
< FinPar ,) >
< IniLla , { >
< cr , - >
< FinLla , } >
< cr , - >
< FinFich , eof >

- Análisis parse

Des 1 2 17 11 6 2 19 22 27 36 40 43 50 45 42 39 6 4 6 2 18 36 40 43 46 53 45 42 38 40 43 50 45 42 39 6 31 21 6 5

Caso 4

/* Programa de ejemplo */
/***** José Luis Fuertes, 5, septiembre, 2015 *********/**
/* El ejemplo incorpora elementos del lenguaje opcionales y elementos que no hay que implementar */

var chars s   /* variable global cadena */

s = "me mola la informatica"

if (False)   {
s="hola"
} else {
s="adios"
}

- Tabla de símbolos
  CONTENIDO DE LA TABLA # TS_General :

  * LEXEMA : s
    ATRIBUTOS :
    + tipo : cadena
    + desplazamiento : 0
    + Tam_parametros : null
    + Tipo_parametro : null
    + Retorno_funcion : null
    + ptrTS_padre : null
  --------- ----------
  ==========

- Tokens

< cr , - >
< cr , - >
< cr , - >
< cr , - >
< PalRes , var >
< PalRes , chars >
< id , s >
< cr , - >
< cr , - >
< id , s >
< OpAsi , = >
< cadena , "me mola la informatica" >
< cr , - >
< cr , - >
< PalRes , if >
< IniPar , (>
< PalRes , False >
< FinPar ,) >
< IniLla , { >
< cr , - >
< id , s >
< OpAsi , = >
< cadena , "hola" >
< cr , - >
< FinLla , } >
< PalRes , else >
< IniLla , { >
< cr , - >
< id , s >
< OpAsi , = >
< cadena , "adios" >
< cr , - >
< FinLla , } >
< cr , - >
< FinFich , eof >

- Análisis parse

Des 1 4 6 4 6 4 6 4 6 2 17 12 6 4 6 2 19 22 27 36 40 43 49 45 42 39 6 4 6 2 18 36 40 43 51 45 42 39 6 30 19 22 27 36 40 43 49 45 42 39 6 31 20 6 30 19 22 27 36 40 43 49 45 42 39 6 31 6 5

Caso 5

```
/* Programa de ejemplo  */
/******* José Luis Fuertes, 5, septiembre, 2015 *********/
/* El ejemplo incorpora elementos del lenguaje opcionales y elementos que no hay que implementar */

var int s        /* variable global cadena */
s = 555 //safafasf

function int FactorialRecursivo (int n, bool p)
{
        if (p &= True)          {
        } else {
        n=s+1
        }
        return FactorialRecursivo ( n , False )
}
var bool b
s = 4
s= 6 + 2
s = FactorialRecursivo ( 8 , True )
```

- Tabla de símbolos
  CONTENIDO DE LA TABLA # TS_General :

  * LEXEMA : s
    ATRIBUTOS :
    + tipo : entero
    + desplazamiento : 0
    + Tam_parametros : null
    + Tipo_parametro : null
    + Retorno_funcion : null
    + ptrTS_padre : null
  --------- ----------
  * LEXEMA : FactorialRecursivo
    ATRIBUTOS :
    + tipo : function
    + desplazamiento : 2
    + Tam_parametros : 2
    + Tipo_parametro : entero,logico
    + Retorno_funcion : entero
    + ptrTS_padre : TS_General
  --------- ----------
  * LEXEMA : b
    ATRIBUTOS :
    + tipo : logico
    + desplazamiento : 6
    + Tam_parametros : null
    + Tipo_parametro : null
    + Retorno_funcion : null
    + ptrTS_padre : null
  --------- ----------
  ==========

CONTENIDO DE LA TABLA # FactorialRecursivo :

* LEXEMA : n
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
* LEXEMA : p
  ATRIBUTOS :
  + tipo : logico
  + desplazamiento : 2
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
==========

- Tokens

< cr , - >
< cr , - >
< cr , - >
< cr , - >
< PalRes , var >
< PalRes , int >
< id , s >
< cr , - >
< id , s >
< OpAsi , = >
< entero , 555 >
< cr , - >
< cr , - >
< PalRes , function >
< PalRes , int >
< id , FactorialRecursivo >
< IniPar , (>
< PalRes , int >
< id , n >
< Coma , , >
< PalRes , bool >
< id , p >
< FinPar ,) >
< cr , - >
< IniLla , { >
< cr , - >
< PalRes , if >
< IniPar , (>
< id , p >
< OpEspcial , &= >
< PalRes , True >
< FinPar ,) >
< IniLla , { >
< cr , - >
< FinLla , } >
< PalRes , else >

< IniLla , { >
< cr , - >
< id , n >
< OpAsi , = >
< id , s >
< Sum , + >
< entero , 1 >
< cr , - >
< FinLla , } >
< cr , - >
< PalRes , return >
< id , FactorialRecursivo >
< IniPar , (>
< id , n >
< Coma , , >
< PalRes , False >
< FinPar ,) >
< cr , - >
< FinLla , } >
< cr , - >
< PalRes , var >
< PalRes , bool >
< id , b >
< cr , - >
< id , s >
< OpAsi , = >
< entero , 4 >
< cr , - >
< id , s >
< OpAsi , = >
< entero , 6 >
< Sum , + >
< entero , 2 >
< cr , - >
< id , s >
< OpAsi , = >
< id , FactorialRecursivo >
< IniPar , (>
< entero , 8 >
< Coma , , >
< PalRes , True >
< FinPar ,) >
< cr , - >
< FinFich , eof >

- Análisis parse

Des 1 4 6 4 6 4 6 4 6 2 17 10 6 2 19 22 27 36 40 43 48 45 42 39 6 4 6 3 7 8 10 13 10 15 11 16 6 6 30 18 36 40 43 46 53 45 42 38 40 43 50 45 42 39 6 31 20 6 30 19 22 27 36 40 43 46 53 44 48 45 42 39 6 31 6 30 19 23 28 36 40 43 46 52 32 36 40 43 46 53 45 42 39 34 36 40 43 51 45 42 39 35 45 42 39 6 31 6 2 17 11 6 2 19 22 27 36 40 43 48 45 42 39 6 2 19 22 27 36 40 43 48 44 48 45 42 39 6 2 19 22 27 36 40 43 46 52 32 36 40 43 48 45 42 39 34 36 40 43 50 45 42 39 35 45 42 39 6 5

Casos de error
Caso 1

var a = 5

- Tokens

< PalRes , var >
< id , a >
< OpAsi , = >
< entero , 5 >
< cr , - >
< cr , - >
< FinFich , eof >

- Error

ERROR Sintactico --> Token_Actual : id en linea :0

- Analisis parse
  Des 1 2 17 12

- Tabla de Simbolos
  CONTENIDO DE LA TABLA # TS_General :

  ==========

Caso 2

var int a
a = "5"
var int b
b = 6

- Tokens

< PalRes , var >
< PalRes , int >
< id , a >
< cr , - >
< id , a >
< OpAsi , = >
< cadena , "5" >
< cr , - >
< PalRes , var >
< PalRes , int >
< id , b >
< cr , - >
< id , b >
< OpAsi , = >
< entero , 6 >
< cr , - >
< cr , - >
< FinFich , eof >

- Error

ERROR Semantico --> asignacion mal formada, en linea: 1

- Analisis parse

Des 1 2 17 10 6 2 19 22 27 36 40 43 49 45 42 39

- Tabla de Simbolos

CONTENIDO DE LA TABLA # TS_General :

* LEXEMA : a
ATRIBUTOS :
+ tipo : entero
+ desplazamiento : 0
+ Tam_parametros : null
+ Tipo_parametro : null
+ Retorno_funcion : null
+ ptrTS_padre : null
--------- ----------
==========

Caso 3

var int n
n = 7

if (n &= True) {}

- Tokens
  < PalRes , var >
  < PalRes , int >
  < id , n >
  < cr , - >
  < id , n >
  < OpAsi , = >
  < entero , 7 >
  < cr , - >
  < cr , - >
  < PalRes , if >
  < IniPar , (>
  < id , n >
  < OpEspcial , &= >
  < PalRes , True >
  < FinPar ,) >
  < IniLla , { >
  < FinLla , } >
  < cr , - >
  < FinFich , eof >

- Error

ERROR Semantico --> Comparacion de elementos con tipos distintos, en linea: 3

- Analisis parse

Des 1 2 17 10 6 2 19 22 27 36 40 43 48 45 42 39 6 4 6 2 18 36 40 43 46 53 45 42 38 40 43 50 45 42 39

- Tabla de Simbolos

CONTENIDO DE LA TABLA # TS_General :

* LEXEMA : n
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 0
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
==========

Caso 4

var bool a
a= true
var int b
b = 8
var int i
if (i = 0)
{
a= a + 1}

- Tokens
  < PalRes , var >
  < PalRes , bool >
  < id , a >
  < cr , - >
  < id , a >
  < OpAsi , = >
  < id , true >
  < cr , - >
  < PalRes , var >
  < PalRes , int >
  < id , b >
  < cr , - >
  < id , b >
  < OpAsi , = >
  < entero , 8 >
  < cr , - >
  < PalRes , var >
  < PalRes , int >
  < id , i >
  < cr , - >
  < PalRes , if >
  < IniPar , (>
  < id , i >
  < OpAsi , = >
  < entero , 0 >
  < FinPar ,) >
  < cr , - >
  < IniLla , { >
  < cr , - >
  < id , a >
  < OpAsi , = >
  < id , a >
  < Sum , + >
  < entero , 1 >
  < FinLla , } >
  < cr , - >
  < FinFich , eof >

- Error

ERROR Semantico --> asignacion mal formada, en linea: 1

- Analisis parse
  Des 1 2 17 11 6 2 19 22 27 36 40 43 46 53 45 42 39

- Tabla de Simbolos

CONTENIDO DE LA TABLA # TS_General :

* LEXEMA : a
  ATRIBUTOS :
  + tipo : logico
  + desplazamiento : 0
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
* LEXEMA : true
  ATRIBUTOS :
  + tipo : entero
  + desplazamiento : 1
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
==========

Caso 5

```
/* Programa de ejemplo  */
/******* José Luis Fuertes, 5, septiembre, 2015 *********/
/* El ejemplo incorpora elementos del lenguaje opcionales y elementos que no hay que implementar */

var int s        /* variable global cadena */
s = 555 //safafasf

function int FactorialRecursivo (int n, bool p)
{
        var chars b
        if (n &= 7)     {
        } else {
        n=s+1
        }
        return FactorialRecursivo ( 8, 7 )
}
var bool b
s = 4
s= 6 + 2
s= FactorialRecursivo ( 8 , "hola" )
```

- Tokens
  < cr , - >
  < cr , - >
  < cr , - >
  < cr , - >
  < PalRes , var >
  < PalRes , int >
  < id , s >
  < cr , - >
  < id , s >

< OpAsi , = >
< entero , 555 >
< cr , - >
< cr , - >
< PalRes , function >
< PalRes , int >
< id , FactorialRecursivo >
< IniPar , (>
< PalRes , int >
< id , n >
< Coma , , >
< PalRes , bool >
< id , p >
< FinPar ,) >
< cr , - >
< IniLla , { >
< cr , - >
< PalRes , var >
< PalRes , chars >
< id , b >
< cr , - >
< PalRes , if >
< IniPar , (>
< id , n >
< OpEspcial , &= >
< entero , 7 >
< FinPar ,) >
< IniLla , { >
< cr , - >
< FinLla , } >
< PalRes , else >
< IniLla , { >
< cr , - >
< id , n >
< OpAsi , = >
< id , s >
< Sum , + >
< entero , 1 >
< cr , - >
< FinLla , } >
< cr , - >
< PalRes , return >
< id , FactorialRecursivo >
< IniPar , (>
< entero , 8 >
< Coma , , >
< entero , 7 >
< FinPar ,) >
< cr , - >
< FinLla , } >
< cr , - >
< PalRes , var >
< PalRes , bool >
< id , b >
< cr , - >
< id , s >
< OpAsi , = >
< entero , 4 >
< cr , - >
< id , s >

< OpAsi , = >
< entero , 6 >
< Sum , + >
< entero , 2 >
< cr , - >
< id , s >
< OpAsi , = >
< id , FactorialRecursivo >
< IniPar , (>
< entero , 8 >
< Coma , , >
< cadena , "hola" >
< FinPar ,) >
< cr , - >
< FinFich , eof >

- Error
  ERROR Semantico --> Parametros mal insertados, son : entero,entero deberia ser : entero,logico, en linea: 14

- Analisis parse
  Des 1 4 6 4 6 4 6 4 6 2 17 10 6 2 19 22 27 36 40 43 48 45 42 39 6 4 6 3 7 8 10 13 10 15 11 16 6 6 30 17 12 6 30 18 36 40 43 46 53 45 42 38 40 43 48 45 42 39 6 31 20 6 30 19 22 27 36 40 43 46 53 44 48 45 42 39 6 31 6 30 19 23 28 36 40 43 46 52 32 36 40 43 48 45 42 39 34 36 40 43 48 45 42 39 35

- Tabla de Simbolos
  CONTENIDO DE LA TABLA # TS_General :

  * LEXEMA : s
    ATRIBUTOS :
    + tipo : entero
    + desplazamiento : 0
    + Tam_parametros : null
    + Tipo_parametro : null
    + Retorno_funcion : null
    + ptrTS_padre : null
  --------- ----------
  * LEXEMA : FactorialRecursivo
    ATRIBUTOS :
    + tipo : function
    + desplazamiento : 2
    + Tam_parametros : 2
    + Tipo_parametro : entero,logico
    + Retorno_funcion : entero
    + ptrTS_padre : TS_General
  --------- ----------
  ==========
  CONTENIDO DE LA TABLA # FactorialRecursivo :

  * LEXEMA : n
    ATRIBUTOS :
    + tipo : entero
    + desplazamiento : 0
    + Tam_parametros : null
    + Tipo_parametro : null
    + Retorno_funcion : null
    + ptrTS_padre : null
  --------- ----------
  * LEXEMA : p

```
  ATRIBUTOS :
  + tipo : logico
  + desplazamiento : 2
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
* LEXEMA : b
  ATRIBUTOS :
  + tipo : cadena
  + desplazamiento : 3
  + Tam_parametros : null
  + Tipo_parametro : null
  + Retorno_funcion : null
  + ptrTS_padre : null
--------- ----------
================================================
```

8. Conclusiones

El trabajo, en su conjunto, ha revestido cierta complejidad, teniendo que destacar especialmente la generación de la gramática. La causa de esta complejidad hay que buscarla en los problemas que surgieron cuando descubrimos que había determinados casos no contemplados o cuando iban apareciendo conflictos que nos encontrábamos en el momento de realizar la tabla.

Otro punto de dificultad fue la elección de un lenguaje concreto para que fuera el soporte sobre el que construir el compilador y llevar adelante la práctica. Elegimos Python por la sencillez tanto en la curva de aprendizaje como en el momento de realizar el desarrollo.

Una vez que conseguimos completar y conocer toda la gramática el proyecto pudo avanzar a mayor velocidad, desapareciendo gran parte de los problemas.

En otro orden de cosas es conveniente resaltar la importancia de las pautas dadas en clase, ya que el seguimiento de las mismas ha permitido y facilitado enormemente la realización del trabajo.

A pesar de todo lo anterior, la práctica nos ha permitido conocer en detalle el funcionamiento interno de un compilador, proporcionándonos un mayor grado de entendimiento de los mecanismos y ciclo de procesos que actúan durante la compilación, en este caso de JavaScript-PL.

Así mismo el trabajo en grupo nos ayuda a comprender la importancia que tiene el trabajo en equipo en los desarrollos y proyectos informáticos. La complejidad del trabajo informático se ve paliada en parte cuando los miembros de un equipo empiezan a trabajar de forma sincronizada y coordinada.

La asignación de tareas, el reparto de cargas, la planificación y sincronización son elementos sin los cuales un trabajo de estas características no podría salir adelante.

Además del trabajo técnico estas facetas han tenido que ser contempladas en este proyecto.