



UNIVERSIDAD POLITÉCNICA DE MADRID

**ESCUELA TÉCNICA SUPERIOR DE INGENIEROS
INFORMÁTICOS**

CURSO 2022 - 2023

“MEMORIA FINAL: PROCESADOR JAVASCRIPT”

AUTORES:

Julio Manso Sánchez-Tornero c200320

Nihel Kella Bouziane c200315

Nº GRUPO:

106

Índice

1. Diseño del Analizador Léxico	3
1.1. Tokens	3
1.2. Gramática	4
1.3. Autómata	5
1.4. Acciones semánticas	6
1.5. Gestión de errores	6
2. Diseño del Analizador Sintáctico	7
2.1. Gramática	7
2.2. Demostración Condición LL1	8
2.3. Procedimientos	9
2.4. Gestión de errores	17
3. Diseño del Analizador Semántico	18
3.1. Traducción Dirigida por la Sintaxis	18
3.2. Gestión de errores	23
4. Diseño de la Tabla de Símbolos	24
4.1. Estructura y organización	24
4.2. Formato	25
ANEXO: Casos prueba	26
Casos de prueba correctos	26
Caso 1	26
Entrada	26
Salida	26
Caso 2	30
Entrada	30
Salida	30
Caso 3	35
Entrada	35
Salida	36
Caso 4	41
Entrada	41
Salida	41
Caso 5	45
Entrada	45
Salida	45
Casos con errores	50
Caso 1	50
Entrada	50
Salida	50
Caso 2	51
Entrada	51
Salida	51

Caso 3	52
Entrada	52
Salida	52
Caso 4	53
Entrada	53
Salida	53
Caso 5	54
Entrada	54
Salida	54

1. Diseño del Analizador Léxico

1.1. Tokens

Se recoge en la siguiente tabla el conjunto de tokens del grupo:

Token	Código	Atributo
let	1	-
int	2	-
string	3	-
boolean	4	-
if	5	-
do	6	-
while	7	-
function	8	-
return	9	-
print	10	-
input	11	-
parizq	12	-
pardrch	13	-
llaveizq	14	-
llavedrch	15	-
puntoYcoma	16	-
coma	17	-
asign	18	-
asignSuma	19	-
mult	20	-
neg	21	-
menor	22	-
cad	23	lexema("c*")
num	24	valor(num)
id	25	posTS(id)

Las opciones obligatorias para el grupo 106 son:

- Sentencia repetitiva (do-while)
- Asignación con suma (+=)
- Comentario de línea (//)
- Comillas simples ('')

Los elementos opcionales seleccionados por el grupo son:

- Operador Aritmético *
- Operador Lógico !
- Operador Relacional <

El atributo- representado en la tabla como '-' indica que se encuentra vacío.

1.2. Gramática

La Gramática regular que representa el lenguaje reconocido por el autómata del Analizador Léxico es la siguiente:

```
● ● ●

1 Gramática del analizador léxico
2
3 0. S -> dA | 1B | _B | 'C | /D | +F | , | ; | delS | ( | ) | { | } | < | ! | =
4 1. A -> dA | λ (oc1)
5 2. B -> 1B | dB | _B | λ (oc2)
6 3. C -> c1C | '
7 4. D -> /E
8 5. E -> c2E | \nS
9 6. F -> =
10
11 1: A-Z, a-z
12 d: 0-9
13 del: espacio, \n, \t, \r
14 c1: todos los caracteres - {''}
15 c2: todos los caracteres - {\n}
16 oc1: otro carácter no dígito
17 oc2: otro carácter ni dígito ni letra ni barra baja
```

Figura 1: Gramática del analizador léxico.

1.3. Autómata

La siguiente figura representa el Autómata Finito Determinista del Analizador Léxico, cuyo estado inicial es el 0 y los estados finales son los representados con dos círculos. Entre cada par de estados existe una transición, en la que se indica el carácter necesario para transitar, así como la acción semántica a realizar (letra azul).

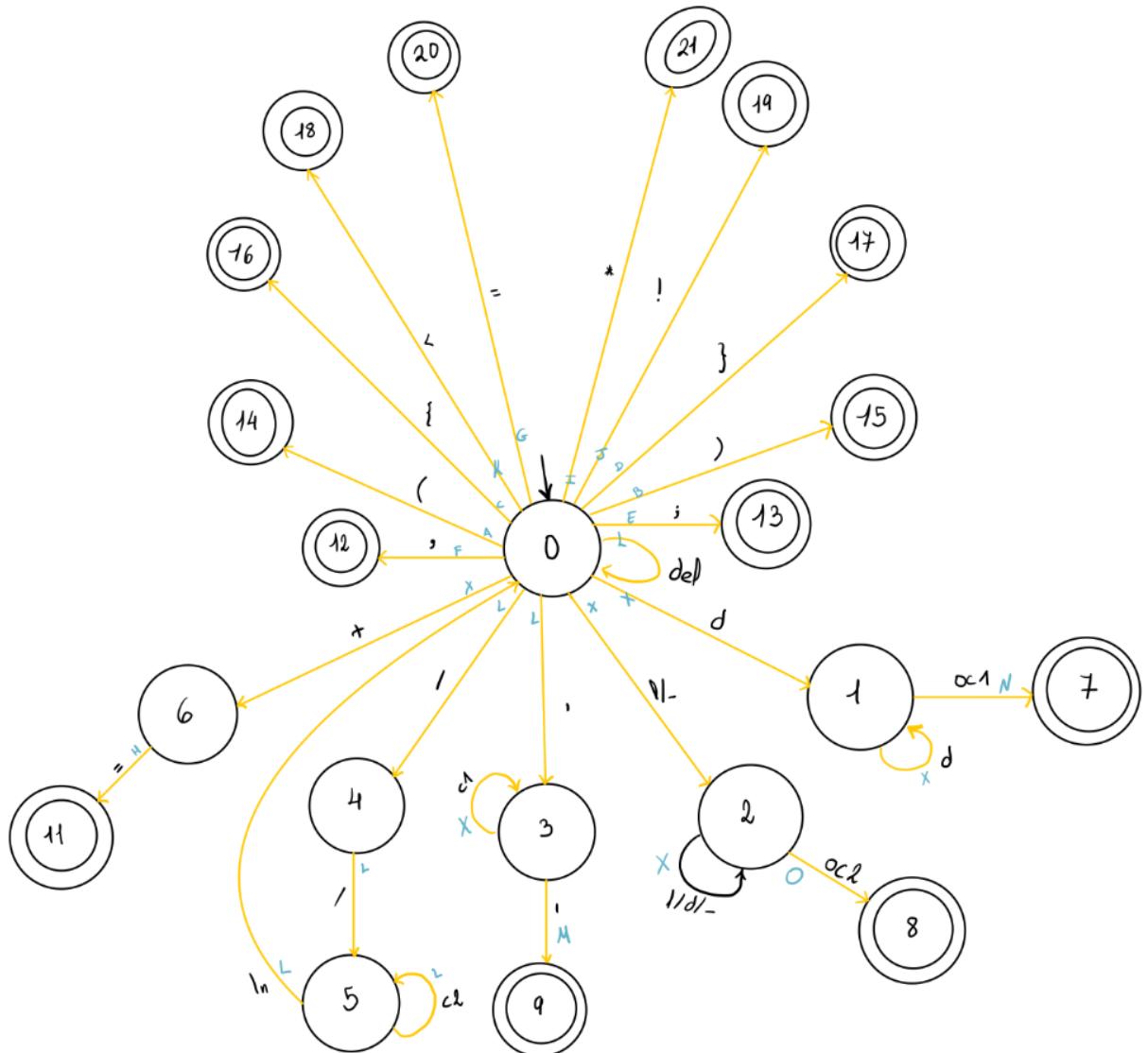


Figura 2: Autómata finito determinista.

1.4. Acciones semánticas

La lectura (L) se realiza en todas las transiciones excepto en las de o_{c1} y o₂, ya que en caso de hacerlo se perdería la información del fichero fuente.

La concatenación (C) se realiza tanto para la formación de cadenas como para la formación de constantes enteras, ya que en la implementación se hará uso de una función de Java de la clase String que convierte una cadena en entero (parseInt(cad)).

```
1 L: Leer -> c = leer()
2 X: Concatenar -> cad = cad + c
3
4 A: GenerarToken(parizq,-)           M: if(caracteres(lexema) > 64) error(2)
5 B: GenerarToken(pardrch,-)          else GenerarToken(cad,lexema)
6 C: GenerarToken(llaveizq,-)         N: if(valor > 32767) error(3)
7 D: GenerarToken(llavedrch,-)        else GenerarToken(num,valor)
8 E: GenerarToken(puntoYcoma,-)       O: if((pos=buscarTPR(lexema)) != -1)
9 F: GenerarToken(coma,-)             GenerarToken(pos,-)
10 G: GenerarToken(asign,-)            else
11 H: GenerarToken(asignSuma,-)       if (!contieneTS(lexema))
12 I: GenerarToken(mult,-)            añadirTS(lexema)
13 J: GenerarToken(neg,-)             posTS = buscarTS(lexema)
14 K: GenerarToken(menor,-)           GenerarToken(id,posTS)
```

Figura 3: Acciones semánticas.

Para facilitar la legibilidad de las acciones semánticas, se ha sustituido el identificador de los tokens por una cadena representativa, es decir, en lugar de parizq en la implementación figuraría el número 12. A la hora de visualizar los tokens en el fichero generado, se añadirá en la misma línea un comentario que especifique el token leído.

1.5. Gestión de errores

En la siguiente tabla se recogen los errores que se pueden producir a nivel léxico:

Código	Error
1	Se ha producido un error en la generación del token. No se esperaba el carácter 'c'.
2	Se ha superado el número máximo de caracteres: 64. Número actual de caracteres: n
3	Entero fuera de rango. El número no debe ser superior a 32767.
4	No se esperaba el carácter '/'. En caso de querer escribir un comentario, solo se admite el siguiente formato: // Comentario
16	Para definir las cadenas de caracteres use comillas simples en vez de dobles.

2. Diseño del Analizador Sintáctico

2.1. Gramática

La Gramática de tipo 2 que representa el lenguaje reconocido por el Analizador Sintáctico está recogida siguiendo el formato de fichero que se indica en la sección de documentación de la asignatura. Como se puede observar, se han numerado las reglas en el orden en el que se emplearán para obtener el parse.

Terminales = {

+ = < * ! id () ent cad = ; } { print input return let if int function boolean string do while , }

NoTerminales = { P E E2 R R2 U V V2 S S2 L Q X B T F H A K C }

Axioma = P

Producciones = {

- | | |
|---------------------|------------------------------------|
| 1. E -> R E2 | 24. Q -> , E Q |
| 2. E2 -> < R E2 | 25. Q -> lambda |
| 3. E2 -> lambda | 26. X -> E |
| 4. R -> U R2 | 27. X -> lambda |
| 5. R2 -> lambda | 28. B -> if (E) S |
| 6. U -> ! V | 29. B -> let id T ; |
| 7. U -> V | 30. B -> S |
| 8. V -> id V2 | 31. B -> do { C } while (E) ; |
| 9. V -> (E) | 32. T -> int |
| 10. V -> ent | 33. T -> boolean |
| 11. V -> cad | 34. T -> string |
| 12. V2 -> (L) | 35. F -> function id H (A) { C } |
| 13. V2 -> lambda | 36. H -> T |
| 14. S -> id S2 | 37. H -> lambda |
| 15. S -> print E ; | 38. A -> T id K |
| 16. R2 -> * U R2 | 39. A -> lambda |
| 17. S -> input id ; | 40. K -> , T id K |
| 18. S -> return X ; | 41. K -> lambda |
| 19. S2 -> = E ; | 42. C -> B C |
| 20. S2 -> (L) ; | 43. C -> lambda |
| 21. S2 -> += E ; | 44. P -> B P |
| 22. L -> E Q | 45. P -> F P |
| 23. L -> lambda | 46. P -> lambda |
- }

2.2. Demostración Condición LL1

A continuación, se muestran los cálculos realizados con el objetivo de demostrar si la gramática cumple la condición LL1. Para ello, antes se han calculado algunos First y Follow para simplificar el desarrollo.

$$\begin{aligned}
 \text{Follow}(E) &= \{ \text{if}, \text{let}, \text{do}, \text{id}, \text{print}, \text{input}, \text{return}, \text{function}, \{, \} \} \\
 \text{FIRST}(Q) &= \{ \text{if}, \text{let}, \text{do}, \text{id}, \text{print}, \text{input}, \text{return}, \text{function}, \{, \} \} \quad \text{Follow}(L) \subset \text{Follow}(E) \\
 \text{Follow}(X) &= \{ \text{if}, \text{let}, \text{do}, \text{id}, \text{print}, \text{input}, \text{return}, \text{function}, \{, \} \} \\
 \text{Follow}(S) &= \text{Follow}(B) \\
 \text{Follow}(B) &= \text{FIRST}(C) \cup \text{FIRST}(P) \cup \text{Follow}(C) \cup \text{Follow}(P) = \{ \text{if}, \text{let}, \text{do}, \text{id}, \text{print}, \text{input}, \text{return}, \text{function}, \{, \} \} \\
 \text{FIRST}(L) &= \text{FIRST}(B) \cap \{ \text{if}, \text{let}, \text{do}, \text{id}, \text{print}, \text{input}, \text{return}, \text{function}, \{, \} \} \quad \text{Follow}(C) \\
 \text{FIRST}(B) &= \{ \text{if}, \text{let}, \text{do}, \text{id}, \text{print}, \text{input}, \text{return} \} \\
 \text{FIRST}(S) &= \{ \text{id}, \text{print}, \text{input}, \text{return} \} \quad \text{Follow}(P) \\
 \text{FIRST}(P) &= \text{FIRST}(B) \cap \text{FIRST}(F) \cap \{ \text{if}, \text{let}, \text{do}, \text{id}, \text{print}, \text{input}, \text{return}, \text{function}, \{, \} \} \\
 \text{FIRST}(F) &= \{ \text{function} \} \\
 \text{Follow}(R) &= \text{FIRST}(E') = \{ \text{if}, \text{let}, \text{do}, \text{id}, \text{print}, \text{input}, \text{return}, \text{function}, \{, \} \} \quad \text{Follow}(E) = \text{Follow}(E') = \{ \text{if}, \text{let}, \text{do}, \text{id}, \text{print}, \text{input}, \text{return}, \text{function}, \{, \} \} \\
 \text{Follow}(V') &= \text{Follow}(V) = \text{Follow}(U) = \text{FIRST}(R') = \{ \text{id}, \text{print}, \text{input}, \text{return} \} \quad \text{Follow}(R) \subset \text{Follow}(V) = \{ \text{id}, \text{print}, \text{input}, \text{return} \} \\
 \text{FIRST}(E) &= \text{FIRST}(RE') = \text{FIRST}(R) = \text{FIRST}(UR') = \text{FIRST}(U) = \{ \text{id}, \text{print}, \text{input}, \text{return} \} \cup \text{FIRST}(V) = \{ \text{id}, \text{print}, \text{input}, \text{return}, \text{ent, cod} \} \\
 \text{FIRST}(V) &= \{ \text{id}, \text{print}, \text{input}, \text{return}, \text{ent, cod} \} \\
 \text{Follow}(R) &= \text{Follow}(R) \\
 \text{Follow}(C) &= \{ \text{id}, \text{print}, \text{input}, \text{return}, \text{ent, cod} \} \\
 \text{Follow}(P) &= \{ \text{id}, \text{print}, \text{input}, \text{return}, \text{ent, cod} \} \\
 \end{aligned}$$

Figura 4: Cálculos previos para demostrar condición LL1.

Evaluación condición 2.1

$$E' \rightarrow \langle RE' \rangle \lambda \rightarrow \text{FIRST}(\langle RE' \rangle) \cap \text{FIRST}(\lambda) = h \in \{ \} \cap \{ \lambda \} = \emptyset$$

$$\hookrightarrow \text{Follow}(E') = \text{Follow}(E) \rightarrow \text{FIRST}(\langle RE' \rangle) \cap \text{Follow}(E') = \emptyset \quad \checkmark$$

$$R' \rightarrow *UR' \lambda \rightarrow \text{FIRST}(*UR') \cap \text{FIRST}(\lambda) = h * h \cap \{ \lambda \} = \emptyset$$

$$\hookrightarrow \text{Follow}(R') = \text{Follow}(R) \rightarrow \text{FIRST}(*UR') \cap \text{Follow}(R) = \emptyset \quad \checkmark$$

$$U \rightarrow !V | V \rightarrow \text{FIRST}(!V) \cap \text{FIRST}(V) = h \{ \} \cap \{ \text{id}, (, \text{ent}, \text{cad} \} = \emptyset \quad \checkmark$$

$$V \rightarrow \text{id}V' | (E) | \text{ent} | \text{cad} \rightarrow \text{FIRST}(\text{id}V') \cap \text{FIRST}(E) \cap \text{FIRST}(\text{ent}) \cap \text{FIRST}(\text{cad}) = \emptyset \quad \checkmark$$

$$V' \rightarrow (L) | \lambda \rightarrow \text{FIRST}((L)) \cap \text{FIRST}(\lambda) = h \{ \} \cap \{ \lambda \} = \emptyset$$

$$\hookrightarrow \text{Follow}(V) = h \{ , \lambda \} \rightarrow \text{FIRST}((L)) \cap \text{Follow}(V) = \emptyset \quad \checkmark$$

$$S \rightarrow E_i | L L; | + = E_j \rightarrow \text{FIRST}(= E_j) \cap \text{FIRST}((L)) \cap \text{FIRST}(+ = E_i) = h = \{ \} \cap h \{ \} \cap h + = \emptyset \quad \checkmark$$

$$L \rightarrow EQ | \lambda \rightarrow \text{FIRST}(EQ) \cap \text{FIRST}(\lambda) = h !, (, \text{id}, \text{ent}, \text{cad} \} \cap \{ \lambda \} = \emptyset$$

$$\hookrightarrow \text{Follow}(L) = h \{ \} \rightarrow \text{FIRST}(EQ) \cap \text{Follow}(L) = \emptyset \quad \checkmark$$

$$Q \rightarrow , EQ | \lambda \rightarrow \text{FIRST}(, EQ) \cap \text{FIRST}(\lambda) = h , \{ \} \cap h \{ \} = \emptyset$$

$$\hookrightarrow \text{Follow}(Q) = \text{Follow}(L) = h \{ \} \rightarrow \text{FIRST}(, EQ) \cap \text{Follow}(L) = \emptyset \quad \checkmark$$

$$S \rightarrow \text{id}S | \text{print}E_i | \text{input}id; | \text{return}X; \rightarrow \text{FIRST}(\text{id}S) \cap \text{FIRST}(\text{print}E_i) \cap \text{First}(\text{input}id;) \cap \text{First}(\text{return}X;) = \emptyset \quad \checkmark$$

$$X \rightarrow E | \lambda \rightarrow \text{FIRST}(E) \cap \text{FIRST}(\lambda) = h !, (, \text{id}, \text{ent}, \text{cad} \} \cap \{ \lambda \} = \emptyset$$

$$\hookrightarrow \text{Follow}(X) = h \{ \} \rightarrow \text{FIRST}(E) \cap \text{Follow}(X) = \emptyset \quad \checkmark$$

$$B \rightarrow \underbrace{\text{if}(E) S}_{1} | \underbrace{\text{let} id T_i}_{2} | \underbrace{S1}_{3} \underbrace{\text{do} h B}_{4} \} \text{while}(E); \rightarrow \text{FIRST}(1) \cap \text{FIRST}(2) \cap \text{FIRST}(3) \cap \text{FIRST}(4) = h \{ \} \cap h \{ \} \cap h \{ \} \cap h \{ \} \cap \{ \text{id}, \text{print}, \text{input}, \text{return} \} \cap h \{ \text{do} h \} = \emptyset \quad \checkmark$$

$$H \rightarrow T | \lambda \rightarrow \text{FIRST}(T) \cap \text{FIRST}(\lambda) = h \{ \text{int}, \text{boolean}, \text{string} \} \cap h \{ \lambda \} = \emptyset$$

$$\hookrightarrow \text{Follow}(H) = h \{ \} \rightarrow \text{FIRST}(T) \cap \text{Follow}(H) = \emptyset \quad \checkmark$$

$$A \rightarrow T ; K | \lambda \rightarrow \text{FIRST}(T ; K) \cap \text{FIRST}(\lambda) = h \{ \text{int}, \text{boolean}, \text{string} \} \cap h \{ \lambda \} = \emptyset$$

$$\hookrightarrow \text{Follow}(A) = h \{ \} \rightarrow \text{FIRST}(T ; K) \cap \text{Follow}(A) = \emptyset \quad \checkmark$$

$$K \rightarrow T ; K | \lambda \rightarrow \text{FIRST}(T ; K) \cap \text{FIRST}(\lambda) = h \{ \} \cap h \{ \lambda \} = \emptyset$$

$$\hookrightarrow \text{Follow}(K) = \text{Follow}(A) = h \{ \} \rightarrow \text{FIRST}(T ; K) \cap \text{Follow}(K) = \emptyset \quad \checkmark$$

$$C \rightarrow BC | \lambda \rightarrow \text{FIRST}(BC) \cap \text{FIRST}(\lambda) = h \{ \text{if}, \text{let}, \text{do}, \text{id}, \text{print}, \text{input}, \text{return} \} \cap h \{ \lambda \} = \emptyset$$

$$\hookrightarrow \text{Follow}(C) = h \{ \} \rightarrow \text{FIRST}(BC) \cap \text{Follow}(C) = \emptyset \quad \checkmark$$

$$P \rightarrow BP | FP | \lambda \rightarrow \text{FIRST}(BP) \cap \text{FIRST}(FP) \cap \text{FIRST}(\lambda) = h \{ \text{if}, \text{let}, \text{do}, \text{id}, \text{print}, \text{input}, \text{return} \} \cap h \{ \text{function} \} \cap h \{ \lambda \} = \emptyset$$

$$\hookrightarrow \text{Follow}(P) = h \{ \} \rightarrow \emptyset \quad \checkmark$$

2.3. Procedimientos

El grupo 106 tiene asignada la técnica de Análisis Descendente recursivo. Se muestran los procedimientos en pseudocódigo. Se designa para la gestión de errores el parámetro cod, que cambiará según las necesidades de la codificación.

```
PROC equipara (Token t){  
    if ( sigToken == t ){  
        sigToken = A_lexico();  
    } else {  
        error(cod);  
    }  
}  
  
PROC E {  
    print(1);  
    R();  
    E2();  
}  
  
PROC E2 {  
    if ( sigToken == '<' ){  
        print(2);  
        equipara('<');  
        R();  
        E2();  
    }else if ( sigToken ∈ { '}' , ';' , ',' }){ // follow(E2)  
        print(3);  
    }else {  
        error(cod);  
    }  
}  
  
PROC R {  
    print(4);  
    U();  
    R2();  
}
```

```

PROC R2 {
    if ( sigToken == '*' ){
        print(5);
        equipara('*');
        U();
        R2();
    }else if ( sigToken ∈ { '<' , '}' , ';' , ',' }){ // follow(R2)
        print(6);
    }else {
        error(cod);
    }
}

```

```

PROC U {
    if ( sigToken == '!' ){
        print(7);
        equipara('!');
        V();
    }else if ( sigToken ∈ { 'id' , '(' , 'ent' , 'cad' }){ // first(V)
        print(8);
        V();
    }else {
        error(cod);
    }
}

```

```

PROC V {
    if ( sigToken == 'id' ){
        print(9);
        equipara('id');
        V2();
    }else if ( sigToken == '(' ){
        print(10);
        equipara( '(' );
        E();
    }
}

```

```

    equipara( ')');

}else if ( sigToken == 'ent' ){

    print(11);

    equipara( 'ent' );

}else if ( sigToken == 'cad' ){

    print(12);

    equipara( 'cad' );

}else {

    error(cod);

}

}

```

```

PROC V2 {

if ( sigToken == '(' ){

    print(13);

    equipara( '(' );

    L();

    equipara( ')' );

}else if ( sigToken ∈ { '*', '<', ')' , ';' , ',' }){ // follow(V2)

    print(14);

}else {

    error(cod);

}

}

```

```

PROC S {

if ( sigToken == 'id' ){

    print(15);

    equipara('id');

    S2();

}else if ( sigToken == 'print' ){

    print(16);

    equipara( 'print' );

    E();

    equipara( ';' );

}else if ( sigToken == 'input' ){

```

```

print(17);
equipara( 'input' );
equipara( 'id' );
equipara( ';' );
}else if ( sigToken == 'return' ){
    print(18);
    equipara( 'return' );
    X();
    equipara( ';' );
}else {
    error(cod);
}
}

```

```

PROC S2 {
    if ( sigToken == '=' ){
        print(19);
        equipara('=');
        E();
        equipara(';');
    }else if ( sigToken == '(' ){
        print(20);
        equipara( '(' );
        L();
        equipara( ')' );
        equipara( ';' );
    }else if ( sigToken == '+=' ){
        print(21);
        equipara( '+=' );
        E();
        equipara( ';' );
    }else {
        error(cod);
    }
}

```

```

PROC L {
    if( sigToken ∈ { ‘!’, ‘id’ , ‘(’ , ‘ent’ , ‘cad’ }){
        print(22);
        E();
        Q();
    }else if ( sigToken == ‘)’ ){
        print(23);
    }else {
        error(cod);
    }
}

PROC Q {
    if( sigToken == ‘,’ ){
        print(24);
        equipara( ‘,’ );
        E();
        Q();
    }else if ( sigToken == ‘)’ ){
        print(25);
    }else {
        error(cod);
    }
}

PROC X {
    if( sigToken ∈ { ‘!’, ‘id’ , ‘(’ , ‘ent’ , ‘cad’ }){
        print(26);
        E();
    }else if ( sigToken == ‘;’ ){
        print(27);
    }else {
        error(cod);
    }
}

```

```

PROC T {
    if ( sigToken == 'int' ){
        print(32);
        equipara('int');
    }else if ( sigToken == 'boolean' ){
        print(33);
        equipara( 'boolean' );
    }else if ( sigToken == 'string' ){
        print(34);
        equipara( 'string' );
    }else {
        error(cod);
    }
}

PROC B {
    if ( sigToken == 'if' ){
        print(28);
        equipara('if');
        equipara( '(' );
        E();
        equipara( ')' );
        S();
    }else if ( sigToken == 'let' ){
        print(29);
        equipara( 'let' );
        equipara( 'id' );
        T();
        equipara( ';' );
    }else if (sigToken ∈ { 'id' , 'print' , 'input' , 'return' }){
        print(30);
        S();
    }else if ( sigToken == 'do' ){
        print(31);
        equipara( 'do' );
        equipara( '{' );
        while( sigToken ∈ { 'if' , 'let' , 'do' , 'id' , 'print' , 'input' , 'return' }){

```

```

C();
}
equipara( '} );
equipara( 'while' );
equipara( '(' );
E();
equipara( ')' );
equipara( ';' );
}else {
    error(cod);
}
}

PROC H {
if( sigToken ∈ { 'int' , 'boolean' , 'string' }){
    print(36);
    T();
}else if ( sigToken == '(' ){ // follow(H)
    print(37);
}else {
    error(cod);
}
}

PROC A {
if( sigToken ∈ { 'int' , 'boolean' , 'string' }){
    print(38);
    T();
    equipara( 'id' );
    K();
}else if ( sigToken == ')' ){ // follow(A)
    print(39);
}else {
    error(cod);
}
}

```

```

PROC K {
    if ( sigToken == ',' ){
        print(40);
        equipara( ',' );
        T();
        equipara(' id ');
        K();
    }else if ( sigToken == ')' ){ // follow(K)
        print(41);
    }else {
        error(cod);
    }
}

```

```

PROC F {
    if ( sigToken == 'function' ){
        print(35);
        equipara('function');
        equipara('id');
        H();
        equipara( '(' );
        A();
        equipara( ')' );
        equipara( '{' );
        while( sigToken ∈ { 'if' , 'let' , 'do' , 'id' , 'print' , 'input' , 'return' } ){
            C();
        }
        equipara( ')' );
    }else {
        error(cod);
    }
}

```

```

PROC P {
    if ( sigToken ∈ { 'if' , 'let' , 'do' , 'id' , 'print' , 'input' , 'return' }){
        print(44);
        B();
        P();
    }else if ( sigToken == 'function' ){
        print(45);
        F();
        P();
    }else if ( sigToken == '$' ){ // follow(P)
        print(46);
    }else {
        error(cod);
    }
}

```

```

PROC C {
    if ( sigToken ∈ { 'if' , 'let' , 'do' , 'id' , 'print' , 'input' , 'return' }){
        print(42);
        B();
        C();
    }else if ( sigToken == '}' ){
        print(43);
    }else {
        error(cod);
    }
}

```

2.4. Gestión de errores

En la siguiente tabla se recogen los errores que se pueden producir a nivel sintáctico:

Código	Error
6	Se ha encontrado 'sigToken' y se esperaba 'token'.
7	Se ha encontrado 'sigToken' y se esperaba uno de los siguientes tokens: 'lista de tokens admitidos'.
8	Se ha encontrado 'sigToken' y se esperaba el fin de fichero.
29	El orden para declarar una variable es: let + identificador + tipo

- *sigToken* corresponde al token devuelto por el analizador léxico.
- *token* corresponde al token que debería haber aparecido.
- *lista de tokens admitidos* corresponde a un enumerado de tokens correctos separados por el carácter ‘|’.

3. Diseño del Analizador Semántico

3.1. Traducción Dirigida por la Sintaxis

Como se ha comentado anteriormente, el grupo 106 tiene asignada la técnica de Análisis Descendente recursivo. A continuación se muestra la traducción dirigida por la sintaxis con las acciones semánticas, asignándose para la gestión de errores el parámetro cod, que cambiará según las necesidades de la codificación.

Inicio -> despGlobal = 0, desplLocal =0, TSG = CrearTSG(), TablaG = true,
zonaDeclaracion = false

P DestruyeTS(TSG)

P-> B P''

```
if(B.tipo == vacío) P.tipo = P''.tipo
else if(B.tipo == tipo_error || P''.tipo == tipo_error) P.tipo=tipo_error
else P.tipo = tipo_ok
```

P -> F P'' if(F.tipo == vacío) P.tipo = P''.tipo
else if (F.tipo == tipo_error || P''.tipo == tipo_error) P.tipo == tipo_error
else P.tipo = tipo_ok }

P -> lambda P.tipo = vacío

B -> if (E) S if(E.tipo != logico) B.tipo = tipo_error
else B.tipo = S.tipo }

B -> let id T ;

```
let zonaDeclaracion = true
zonaDeclaracion = false
id if(tablaG) // esta activa TSG
InsertarTipoTS(id.pos, T.tipo, tablaSimGlobal)
InsertarDespTS(id.pos, despl, tablaSimGlobal)
despGlobal = desplGlobal + T.ancho
else
InsertarTipoTS(id.pos, T.tipo, tablaSimLocal)
InsertarDespTS(id.pos, despl, tablaSimLocall)
desplLocal = desplLocal + T.ancho
B.tipo = tipo_ok
```

B -> S B.tipo = S.tipo

B -> do { C } B.tipo = C.tipo
 while (E); if(E.tipo != logico) B.tipo = tipo_error

C -> B C'' if(B.tipo == tipo_error) C.tipo = tipo_error
 if(C''.tipo == vacio) C.tipo = tipo_ok
 else C.tipo = C''.tipo

C -> lambda C.tipo = vacío

T-> int T.tipo = ent T.ancho = 1

T-> boolean T.tipo = lógico T.ancho = 1

T-> string T.tipo = cad T.ancho = 64

E -> R E2
 if(R.tipo == ent && E2.tipo == logico) E.tipo = logico
 else E.tipo = R.tipo

S -> id S2 pos = buscaIdTS(id)
 if(pos == -1){
 insertoTipoTSG(id.pos, ent) // tipo entero
 insertoDesplTSG(id.pos, desplGlobal)
 desplGlobal = desplGlobal +1
 }
 idTipo = buscaTipoTS(pos)
 if(idTipo == fun){
 numParam = NumParamTS(pos)
 tipoParam = TipoParamTS(pos)
 if(S2.numParam == numParam && S2.tipoParam == tipoParam)
 S.tipo = tipo_ok
 else
 S.tipo = tipo_error
 }
 else if(idTipo == S2.tipo) S.tipo = tipo_ok

S -> print E ; if(E.tipo == cad || E.tipo == ent)
 S.tipo = tipo_ok
 else S.tipo = tipo_error

S -> input id ;

```

pos = buscaIdTS(id)
if(pos == -1) {
    insertoTipoTSG(id.pos, ent) // tipo entero
    insertoDesplTSG(id.pos, desplGlobal)
    desplGlobal = desplGlobal +1
}
idTipo = buscaTipoTS(pos)
if (idTipo == cad || idTipo == ent) S.tipo = tipo_ok
else S.tipo = tipo_error
}

S -> return X ; if(!tablaG){
    tipoRetorno = buscaTipoRetornoTS(funActual)
    if(X.tipo == tipoRetorno) S.tipo = tipo_ok, S.tipoRetorno = tipoRetorno
    else S.tipo = tipo_error, S.tipoRetorno = tipo_error
}
else S.tipo = tipo_ok, S.tipoRetorno = X.tipo}

.....
E2-> < R E2'   if(R.tipo = ent && (E2.tipo = ent || E2.tipo == vacio))
        E2.tipo = lógico else error}

E2 -> lambda      E2.tipo = vacío
.....
R -> U R2      if(R2.tipo == ent && U.tipo == ent)
        R.tipo = ent
        else R.tipo = U.tipo
.....
R2 -> * U R2"  if(U.tipo == ent && (R2".tipo == ent || R2".tipo == vacio))
        R2.tipo = ent
        else R2.tipo = tipo_error

R2 -> lambda      R2.tipo = vacío
.....
U -> ! V      if(V.tipo == lógico)
        U.tipo = lógico
        else U.tipo = tipo_error

U -> V      U.tipo = V.tipo
.....
V -> ( E )      V.tipo = E.tipo

V -> id V2      pos = buscaIdTS(id)
if(pos == NULL) {
    V.tipo = tipo_error
}
idTipo = buscaTipoTS(pos)
if(idTipo == fun){

```

```

if(V2.tipo == vacío) V2.tipo = tipo_error
else{
    tipoParam = buscaTipoParamTSG(id.pos)
    numParam = buscaNumParamTSG(id.pos)
    tipoRetorno = buscaTipoRetTSG(id.pos)
    if(V2.numParam == numParam && V2.tipoParam ==
       tipoParam){
        V.tipo = tipoRetorno
    }
}
}
else {V.tipo = id.tipo}

```

V -> ent V.tipo = ent

V -> cad V.tipo = cad

S2 -> = E ; S2.tipo = E.tipo

S2 -> (L) ; S2.tipo = fun
 S2.numParam = L.numParam
 S2.tipoParam = L.tipoParam

S2 -> += E ; S2.tipo = E.tipo

V2 -> lambda V2.tipoParam = vacío
 V2.numParam = 0

V2 -> (L) V2.tipoParam = L.tipoParam
 V2.numParam = L.numParam

F -> function id H

```

zonaDecl = true
TSL = crearTSL()
TSactual = TSL
funActual = id
desplLocal = 0
insertarTipoTS(id.pos, fun, tablaSimGlobal)
insertarEtiqTSG(id.pos, nuevaEtiquetaTS)
tipoRetorno = H.tipo
insertarTipoRetornoTS(id.pos, tipoRetorno, tablaSimGlobal)

```

(A) insertarTipoParamTS(funActual.pos, A.tipoParam)
 insertarNParamTSG(funActual.Pos, A.numParam)
 zonaDecl = false
 if(A.tipo == tipo_error){
 F.tipo = tipo_error
 }

```

{ C }      if(!H.tipo == vacio && !tieneRetorno)
            F.tipo = tipo_error
else if ( C.tipo == vacio && !tipoRetorno == vacío){
            F.tipo = tipo_error
}else  F.tipo = tipo_ok
destruirTSL()

-----
L -> E Q  if(Q.tipo == vacío) {
                L.tipoParam = E.tipo
                L.numParam = 1
}
else
                L.tipoParam = E.tipo y Q.tipoParam
                L.numParam = Q.numParam + 1
}
L -> lambda  L.tipo = vacío
                L.numParam = 0

-----
Q -> , E Q'' if(Q.tipo == vacío) {
                Q.tipoParam = E.tipo
                Q.numParam = 1
}
else{
                Q.tipoParam = E.tipo y Q''.tipoParam
                Q.numParam = Q''.numParam + 1
}

Q -> lambda  Q.tipo = vacío

-----
H -> T          H.tipo = T.tipo

H -> lambda  H.tipo = vacío

-----
A -> T id K    insertarTipoTS(id.pos, T.tipo, tablaSimLocal),
                    insertarDespTS(id.pos, despLocal, tablaSimLocal),
                    despLocal = despLocal + T.ancho
if(K.tipo == vacio) {
            A.tipo = T.tipo
            A.numParam = 1
}
else {
            A.tipoParam = T.tipo y K.tipoParam,
            A.numParam = K.numParam + 1
}

```

A -> lambda	A.tipo = vacío
-----------------------	----------------

K -> , T id K''	insertarTipoTS(size(TSL)-1, T.tipo, tablaSimLocal), insertarDespTS(size(TSL)-1, desplLocal, tablaSimLocal), desplLocal = desplLocal + T.ancho if(K.tipo = vacio) { K.tipoParam = T.tipo, K.numParam = 1 } else { K.tipoParam = T.tipo y K''.tipoParam K.numParam = K''.numParam + 1 }
---------------------------	--

K -> lambda	K.tipo = vacío
-----------------------	----------------

X -> E	X.tipo = E.tipo
X -> lambda	X.tipo = vacío

3.2. Gestión de errores

Código	Error
5	Ya existe el identificador 'id'. Elija otro nombre.
9	El identificador 'id' no está declarado.
10	La expresión debe ser de tipo lógico.
11	Se ha encontrado un parámetro de tipo 'tipo1' y se esperaba un parámetro de tipo 'tipo2'.
12	La instrucción print solo evalua expresiones de tipo cadena o de tipo entero.
13	No se ha definido un nombre para la función.
14	La función no tiene definido un valor de retorno.
15	Ambos lados de la expresión deben de ser de tipo entero.
17	La función 'id' no está declarada.
18	El lenguaje no permite la definición de funciones anidadas.
19	El lenguaje no permite la definición de funciones dentro de un bucle.
20	El tipo de retorno ('tipo1') no coincide con el definido en la función ('tipo2').

21	La función tiene que devolver una expresión de tipo 'tipo'.
22	Una función sin retorno no debe devolver nada.
23	Ambos lados de la expresión deben de ser de tipo 'tipo'.
24	Los operadores aritméticos (+=) solo se aplican sobre tipos enteros.
25	Los operadores relacionales (<) solo se aplican sobre tipos enteros.
26	Los operadores lógicos (!) solo se aplican sobre tipos lógicos.
27	Los operadores aritméticos (*) solo se aplican sobre tipos enteros.
28	No coincide el número de parámetros de la función. Debería tener 'n' parámetros.
30	La función 'id' debe tener retorno al no ser de tipo vacío.
31	La instrucción input solo evalua expresiones de tipo cadena o de tipo entero.
32	No se puede asignar un valor a una función.

4. Diseño de la Tabla de Símbolos

4.1. Estructura y organización

Para la tabla de símbolos global se ha empleado la estructura de datos ArrayList (**tablaSimGlobal**) en la que se introduce, para cada identificador declarado, un ArrayList (**atributos**). De modo que, en la práctica será un ArrayList de ArrayList. El más interno de ellos, contendrá los siguientes atributos:

- Lexema.
- Tipo.
- Desplazamiento.
- Número de Parámetros.
- Tipo de Parámetros.
- Tipo devuelto.
- Etiqueta.

Los identificadores de funciones son los únicos que tendrán definidos el número de parámetros, tipo de parámetros, tipo devuelto y etiqueta, puesto que solo para ellos son relevantes. Sin embargo, el atributo 'desplazamiento' permanecerá vacío para este tipo de dato.

En la tabla de símbolos global introducimos las variables declaradas fuera de las funciones (let id tipo;) o las variables no declaradas de forma explícita, ya sea dentro o fuera de las funciones.

En cuanto a la tabla de símbolos local de cada una de las funciones que tengamos en código, procede de la siguiente manera:

- Un ArrayList (**tablasLocales**), en el que tendremos un ArrayList para cada una de las funciones que existan. Será donde se almacenarán todas las tablas locales.
- Un ArrayList (**tablaSimLocal**) en el que tendremos un ArrayList (**atributos**) para cada uno de las variables que tenga la función, ya sean parámetros o sean variables declaradas dentro de manera no implícita. Estas variables solo tendrán 3 atributos (lexema, tipo y desplazamiento).
- Un ArrayList (**nombreLocales**) en el que tendremos el nombre de cada una de las funciones. Su posición será equivalente a la de *tablasLocales* por lo que así podremos identificar cada tabla local con su respectiva función.

Por otro lado, y lo más relevante de la organización, tanto para la tabla de símbolos global como para la local tendremos un mapa asociado (**mapaTSG** para la global y **mapaTSL** para cada local). En estos mapas tendremos, para cada variable, su lexema (clave) y posición (valor) en su respectiva tabla de símbolos. De esta manera, se consigue una mayor eficiencia en la búsqueda, puesto que en ArrayList tiene un coste mucho mayor.

4.2. Formato

El formato de la tabla de símbolos será el siguiente y se repetirá tantas veces como identificadores aparezcan en el fichero fuente:

CONTENIDO DE LA TABLA DE SÍMBOLOS (visibilidad) #nº tabla :
* LEXEMA : 'nombreID'
ATRIBUTOS:
+ tipo: 'tipoid'
+ despl: despId
----- // Separación entre lexemas
----- // Separación entre tablas

visibilidad = GLOBAL o LOCAL

nº tabla = 1 (si *visibilidad* = GLOBAL) o [2, ...] (si *visibilidad* = LOCAL)

En cuanto a los identificadores de tipo función contenidos en la tabla de símbolos global, el formato será el siguiente (se omiten separadores):

* LEXEMA : 'nombreID'
ATRIBUTOS:
+ tipo: 'tipoid'
+ numParam: nºparamId
+ TipoParamX: 'TipoParamId'
+ TipoRetorno: 'TipoRetid'
+ EtiqFuncion: 'EtiqFunY'

X = será el número de parámetro de la función ($X \leq \text{numParam}$) .

Y = será el número de función.

ANEXO: Casos prueba

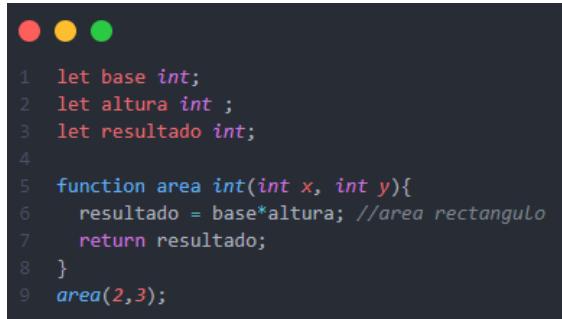
Se incluyen 10 casos prueba listados de los cuales la mitad de ellos serán correctos y la otra mitad serán erróneos, de tal manera que permitan observar el comportamiento del Procesador. Para los ejemplos correctos se incluirá el listado de tokens, el árbol de análisis sintáctico (que se generará utilizando la herramienta VASt) y el volcado de la Tabla de Símbolos. Para los 5 ejemplos erróneos se incluirá el mensaje de error obtenido.

Casos de prueba correctos

En primer lugar, se proponen una serie de casos prueba en los que no hay errores semánticos, por lo que se deberían obtener los volcados de manera correcta.

Caso 1

Entrada



```
let base int;
let altura int ;
let resultado int;
function area int(int x, int y){
    resultado = base*altura; //area rectangulo
    return resultado;
}
area(2,3);
```

Figura 5: Fichero de entrada *Area.js*

Salida



1 <1, > // token palabra reservada let	21 <25,1> // token identificador y
2 <25,0> // token identificador base	22 <13, > // token parDrch
3 <2, > // token palabra reservada int	23 <14, > // token llaveIzq
4 <16, > // token puntoYcoma	24 <25,2> // token identificador resultado
5 <1, > // token palabra reservada let	25 <18, > // token asign
6 <25,1> // token identificador altura	26 <25,0> // token identificador base
7 <2, > // token palabra reservada int	27 <20, > // token mult
8 <16, > // token puntoYcoma	28 <25,1> // token identificador altura
9 <1, > // token palabra reservada let	29 <16, > // token puntoYcoma
10 <25,2> // token identificador resultado	30 <9, > // token palabra reservada return
11 <2, > // token palabra reservada int	31 <25,2> // token identificador resultado
12 <16, > // token puntoYcoma	32 <16, > // token puntoYcoma
13 <8, > // token palabra reservada function	33 <15, > // token llaveDrch
14 <25,3> // token identificador area	34 <25,3> // token identificador area
15 <2, > // token palabra reservada int	35 <12, > // token parIzq
16 <12, > // token parIzq	36 <24,2> // token constante entera
17 <2, > // token palabra reservada int	37 <17, > // token coma
18 <25,0> // token identificador x	38 <24,3> // token constante entera
19 <17, > // token coma	39 <13, > // token parDrch
20 <2, > // token palabra reservada int	40 <16, > // token puntoYcom

Figura 6: Fichero de tokens *Area.js*

```
● ○ ●  
1 Descendente 44 29 32 44 29 32 44 29 32 45 35 36 32 38 32 40  
2 32 41 42 30 15 19 1 4 8 9 14 5 8 9 14 6 3 42 30 18 26 1 4 8  
3 9 14 6 3 43 44 30 15 20 22 1 4 8 11 6 3 24 1 4 8 11 6 3 25  
4 46
```

Figura 7: Fichero de parse *Area.js*

```
● ○ ●  
1 CONTENIDO DE LA TABLA DE SIMBOLOS GLOBAL #1 :  
2  
3 * LEXEMA : 'base'  
4 ATRIBUTOS:  
5 + tipo: 'entero'  
6 + despl: 0  
7 -----  
8 * LEXEMA : 'altura'  
9 ATRIBUTOS:  
10 + tipo: 'entero'  
11 + despl: 1  
12 -----  
13 * LEXEMA : 'resultado'  
14 ATRIBUTOS:  
15 + tipo: 'entero'  
16 + despl: 2  
17 -----  
18 * LEXEMA : 'area'  
19 ATRIBUTOS:  
20 + tipo: 'funcion'  
21 + numParam: 2  
22 + TipoParam1: 'entero'  
23 + TipoParam2: 'entero'  
24 + TipoRetorno: 'entero'  
25 + EtiqFuncion: 'EtFun1'  
26 -----  
27 -----  
  
● ○ ●  
1 CONTENIDO DE LA TABLA DE SIMBOLOS LOCAL DE LA FUNCION area #2 :  
2  
3 * LEXEMA : 'x'  
4 ATRIBUTOS:  
5 + tipo: 'entero'  
6 + despl: 0  
7 -----  
8 * LEXEMA : 'y'  
9 ATRIBUTOS:  
10 + tipo: 'entero'  
11 + despl: 1  
12 -----  
13 -----
```

Figura 8: Fichero de TS *Area.js*

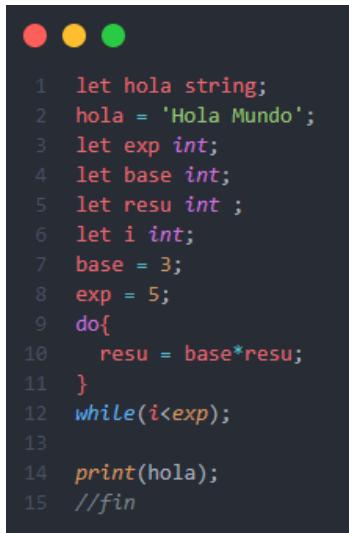




Figura 9: Fichero del árbol *Area.js*

Caso 2

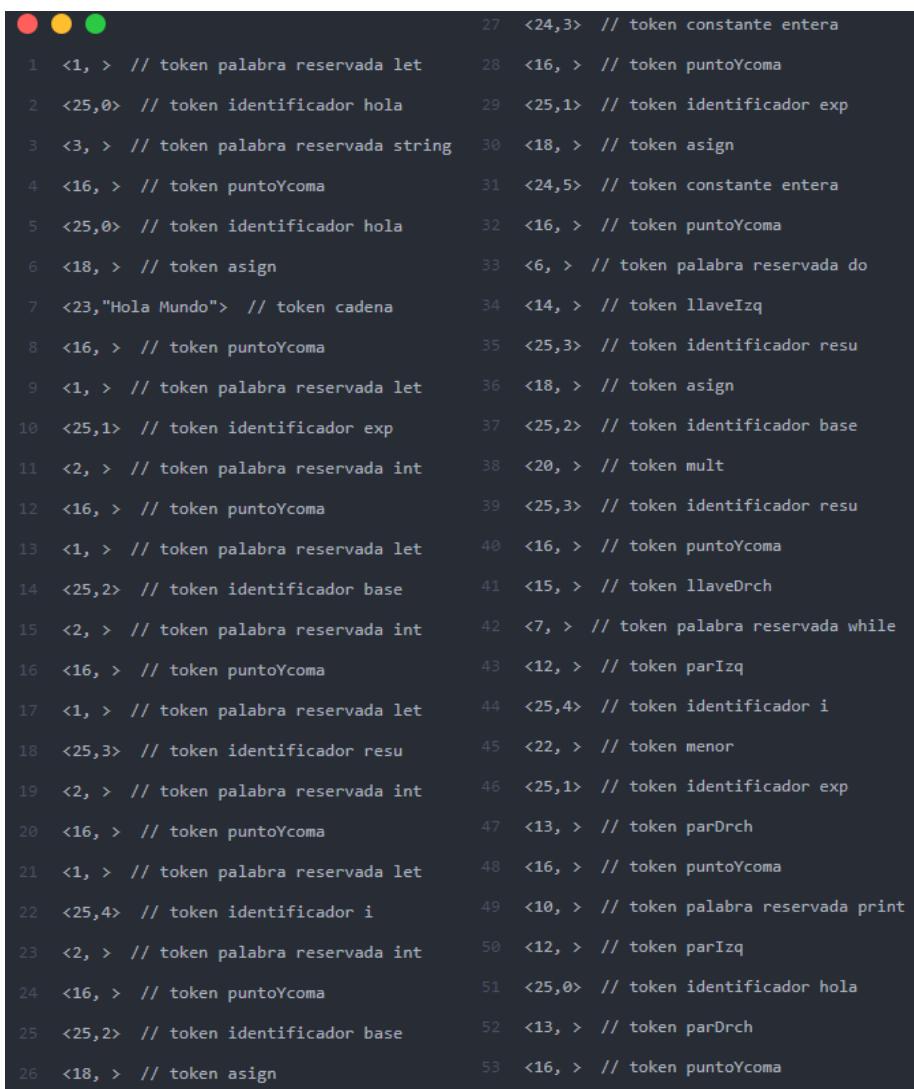
Entrada



```
1 let hola string;
2 hola = 'Hola Mundo';
3 let exp int;
4 let base int;
5 let resu int ;
6 let i int;
7 base = 3;
8 exp = 5;
9 do{
10     resu = base*resu;
11 }
12 while(i<exp);
13
14 print(hola);
15 //fin
```

Figura 10: Fichero de entrada *AreaTriangulo.js*

Salida



```
1 <1, > // token palabra reservada let           27 <24,3> // token constante entera
2 <25,0> // token identificador hola          28 <16, > // token puntoYcoma
3 <3, > // token palabra reservada string        29 <25,1> // token identificador exp
4 <16, > // token puntoYcoma                  30 <18, > // token asign
5 <25,0> // token identificador hola          31 <24,5> // token constante entera
6 <18, > // token asign                         32 <16, > // token puntoYcoma
7 <23,"Hola Mundo"> // token cadena          33 <6, > // token palabra reservada do
8 <16, > // token puntoYcoma                  34 <14, > // token llaveIzq
9 <1, > // token palabra reservada let          35 <25,3> // token identificador resu
10 <25,1> // token identificador exp           36 <18, > // token asign
11 <2, > // token palabra reservada int          37 <25,2> // token identificador base
12 <16, > // token puntoYcoma                  38 <20, > // token mult
13 <1, > // token palabra reservada let          39 <25,3> // token identificador resu
14 <25,2> // token identificador base          40 <16, > // token puntoYcoma
15 <2, > // token palabra reservada int          41 <15, > // token llaveDrch
16 <16, > // token puntoYcoma                  42 <7, > // token palabra reservada while
17 <1, > // token palabra reservada let          43 <12, > // token parIzq
18 <25,3> // token identificador resu          44 <25,4> // token identificador i
19 <2, > // token palabra reservada int          45 <22, > // token menor
20 <16, > // token puntoYcoma                  46 <25,1> // token identificador exp
21 <1, > // token palabra reservada let          47 <13, > // token parDrch
22 <25,4> // token identificador i             48 <16, > // token puntoYcoma
23 <2, > // token palabra reservada int          49 <10, > // token palabra reservada print
24 <16, > // token puntoYcoma                  50 <12, > // token parIzq
25 <25,2> // token identificador base          51 <25,0> // token identificador hola
26 <18, > // token asign                         52 <13, > // token parDrch
27 <16, > // token puntoYcoma
```

Figura 11: Fichero de tokens *AreaTriangulo.js*

```
1 Descendente 44 29 34 44 30 15 19 1 4 8 12 6 3 44 29 32 44 29 32 44  
2 29 32 44 29 32 44 30 15 19 1 4 8 11 6 3 44 30 15 19 1 4 8 11 6 3  
3 44 31 42 30 15 19 1 4 8 9 14 5 8 9 14 6 3 43 1 4 8 9 14 6 2 4 8  
4 9 14 6 3 44 30 16 1 4 8 10 1 4 8 9 14 6 3 6 3 46
```

Figura 12: Fichero de parse AreaTriangulo.js

```
1 CONTENIDO DE LA TABLA DE SIMBOLOS GLOBAL #1 :  
2  
3 * LEXEMA : 'hola'  
4 ATRIBUTOS:  
5 + tipo: 'cadena'  
6 + despl: 0  
7 -----  
8 * LEXEMA : 'exp'  
9 ATRIBUTOS:  
10 + tipo: 'entero'  
11 + despl: 64  
12 -----  
13 * LEXEMA : 'base'  
14 ATRIBUTOS:  
15 + tipo: 'entero'  
16 + despl: 65  
17 -----  
18 * LEXEMA : 'resu'  
19 ATRIBUTOS:  
20 + tipo: 'entero'  
21 + despl: 66  
22 -----  
23 * LEXEMA : 'i'  
24 ATRIBUTOS:  
25 + tipo: 'entero'  
26 + despl: 67  
27 -----
```

Figura 13: Fichero de TS AreaTriangulo.js

- P (44)
 - B (29)
 - let
 - id
 - T (34)
 - string
 - ;
 - P (44)
 - B (30)
 - S (15)
 - id
 - S2 (19)
 - =
 - E (1)
 - R (4)
 - U (8)
 - V (12)
 - cad
 - R2 (6)
 - lambda
 - E2 (3)
 - lambda
 - ;
 - P (44)
 - B (29)
 - let
 - id
 - T (32)
 - int
 - ;
 - P (44)
 - B (29)
 - let
 - id
 - T (32)
 - int
 - ;
 - P (44)
 - B (29)
 - let
 - id
 - T (32)
 - int
 - ;
 - P (44)
 - B (29)
 - let
 - id
 - T (32)
 - int
 - ;
 - P (44)
 - B (30)
 - S (15)



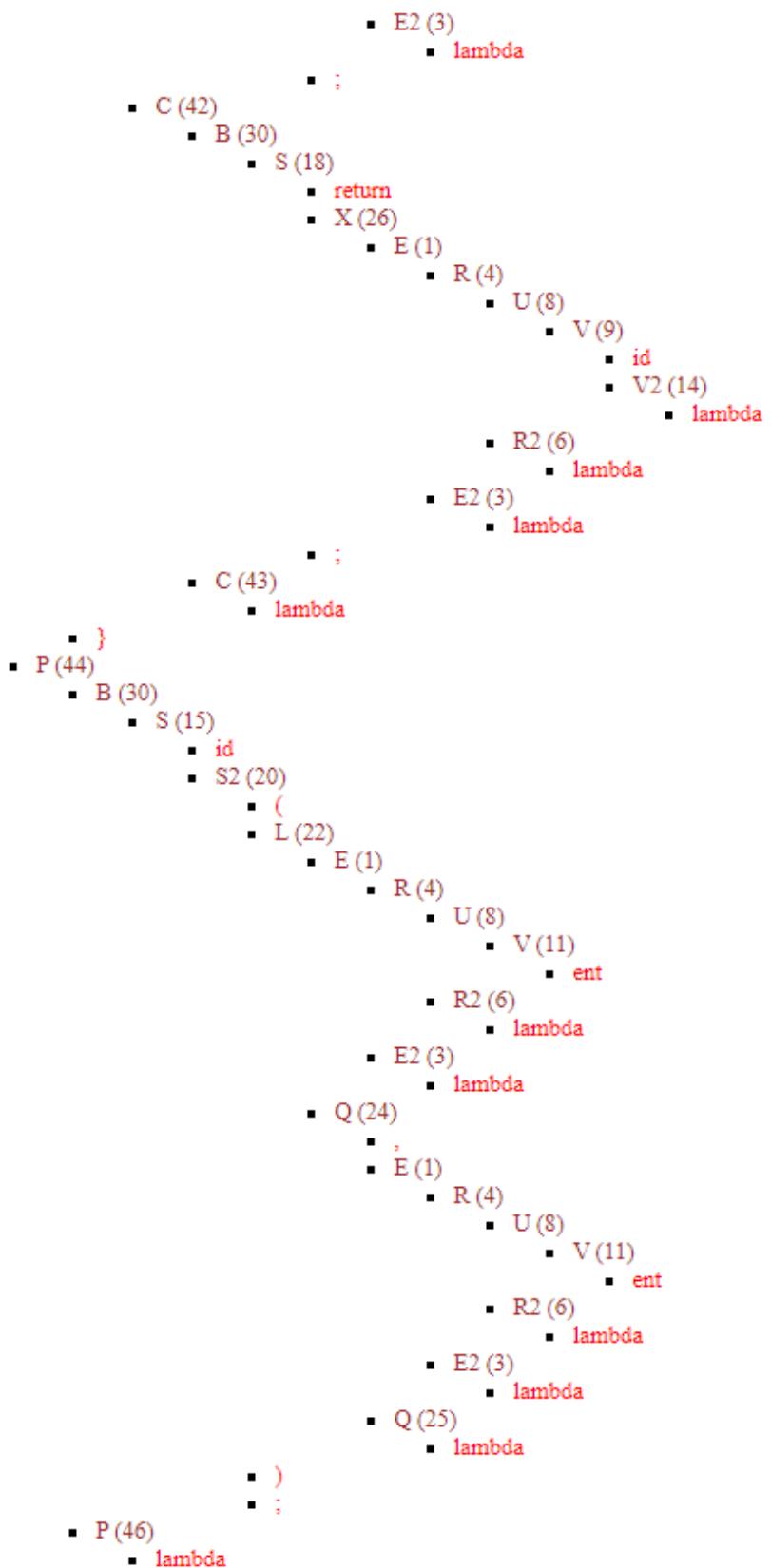


Figura 14: Fichero del árbol *AreaTriangulo.js*

Caso 3

Entrada



```
1 let barraPan string;
2 let aguacate string;
3 let pomelo string;
4 let amasado boolean;
5
6 function amasador (string cosa, string cosa2){
7     do{
8         barraPan = 'amasando';
9     }
10    while(!amasado);
11    pomelo = aguacate;
12 }
13
14 function hornear boolean (string barraPan){
15     return amasado;
16 }
```

Figura 15: Fichero de entrada *Panaderia.js*

Salida

```
1 <1, > // token palabra reservada let           29 <25,0> // token identificador barraPan
2 <25,0> // token identificador barraPan       30 <18, > // token asign
3 <3, > // token palabra reservada string        31 <23,"amasando"> // token cadena
4 <16, > // token puntoYcoma                   32 <16, > // token puntoYcoma
5 <1, > // token palabra reservada let           33 <15, > // token llaveDrch
6 <25,1> // token identificador aguacate      34 <7, > // token palabra reservada while
7 <3, > // token palabra reservada string        35 <12, > // token parIzq
8 <16, > // token puntoYcoma                   36 <21, > // token neg
9 <1, > // token palabra reservada let           37 <25,3> // token identificador amasado
10 <25,2> // token identificador pomelo        38 <13, > // token parDrch
11 <3, > // token palabra reservada string        39 <16, > // token puntoYcoma
12 <16, > // token puntoYcoma                   40 <25,2> // token identificador pomelo
13 <1, > // token palabra reservada let           41 <18, > // token asign
14 <25,3> // token identificador amasado        42 <25,1> // token identificador aguacate
15 <4, > // token palabra reservada boolean       43 <16, > // token puntoYcoma
16 <16, > // token puntoYcoma                   44 <15, > // token llaveDrch
17 <8, > // token palabra reservada function      45 <8, > // token palabra reservada function
18 <25,4> // token identificador amasador        46 <25,5> // token identificador hornear
19 <12, > // token parIzq                         47 <4, > // token palabra reservada boolean
20 <3, > // token palabra reservada string        48 <12, > // token parIzq
21 <25,0> // token identificador cosa            49 <3, > // token palabra reservada string
22 <17, > // token coma                           50 <25,0> // token identificador barraPan
23 <3, > // token palabra reservada string        51 <13, > // token parDrch
24 <25,1> // token identificador cosa2          52 <14, > // token llaveIzq
25 <13, > // token parDrch                      53 <9, > // token palabra reservada return
26 <14, > // token llaveIzq                     54 <25,3> // token identificador amasado
27 <6, > // token palabra reservada do           55 <16, > // token puntoYcoma
28 <14, > // token llaveIzq                     56 <15, > // token llaveDrch
```

Figura 16: Fichero de tokens *Panaderiajs*



```

1 Descendente 44 29 34 44 29 34 44 29 34 44 29 33 45 35 37 38
2 34 40 34 41 42 31 42 30 15 19 1 4 8 12 6 3 43 1 4 7 9 14 6 3
3 42 30 15 19 1 4 8 9 14 6 3 43 45 35 36 33 38 34 41 42 30 18 26
4 1 4 8 9 14 6 3 43 46

```

Figura 17: Fichero de parse *Panaderia.js*



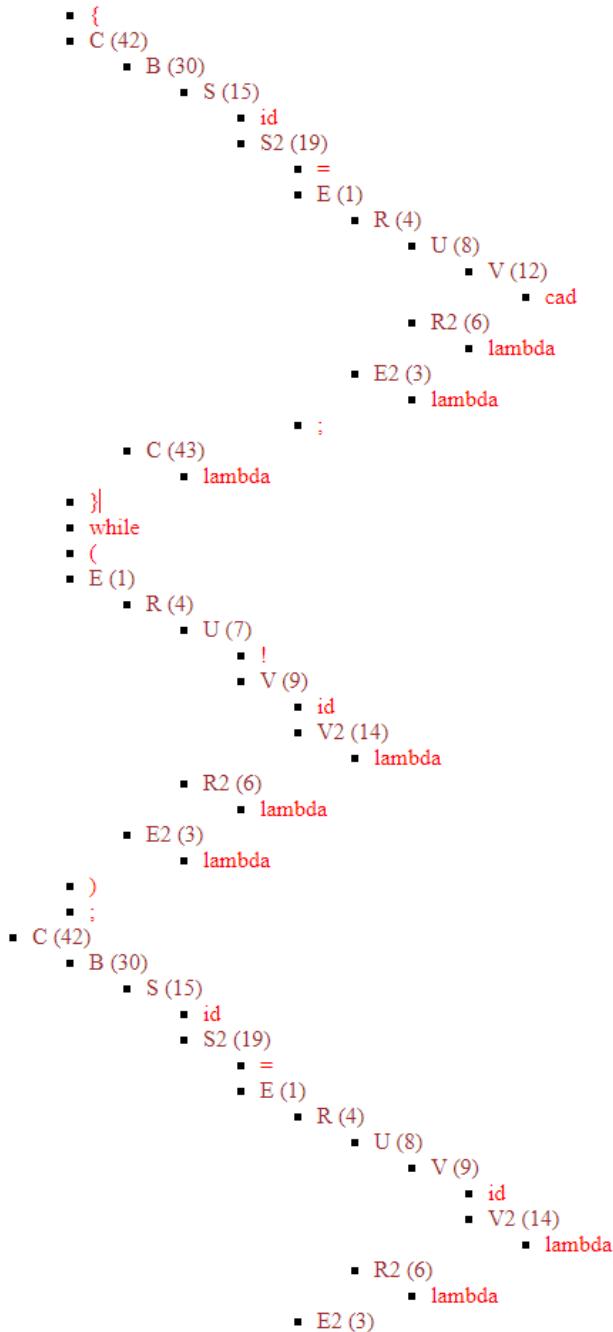
```

1 CONTENIDO DE LA TABLA DE SIMBOLOS GLOBAL #1 :
2
3 * LEXEMA : 'barraPan'
4 ATRIBUTOS:
5 + tipo: 'cadena'
6 + despl: 0
7 -----
8 * LEXEMA : 'aguacate'
9 ATRIBUTOS:
10 + tipo: 'cadena'
11 + despl: 64
12 -----
13 * LEXEMA : 'pomelo'
14 ATRIBUTOS:
15 + tipo: 'cadena'
16 + despl: 128
17 -----
18 * LEXEMA : 'amasado'
19 ATRIBUTOS:
20 + tipo: 'logico'
21 + despl: 192
22 -----
23 * LEXEMA : 'amasador'
24 ATRIBUTOS:
25 + tipo: 'funcion'
26 + numParam: 2
27 + TipoParam1: 'cadena'
28 + TipoParam2: 'cadena'
29 + TipoRetorno: 'vacio'
30 + EtiqFuncion: 'EtFun1'
31 -----
32 * LEXEMA : 'hornear'
33 ATRIBUTOS:
34 + tipo: 'funcion'
35 + numParam: 1
36 + TipoParam1: 'cadena'
37 + TipoRetorno: 'logico'
38 + EtiqFuncion: 'EtFun2'
39 -----
40 -----
41 CONTENIDO DE LA TABLA DE SIMBOLOS LOCAL DE LA FUNCION amasador #2 :
42
43 * LEXEMA : 'cosa'
44 ATRIBUTOS:
45 + tipo: 'cadena'
46 + despl: 0
47 -----
48 * LEXEMA : 'cosa2'
49 ATRIBUTOS:
50 + tipo: 'cadena'
51 + despl: 64
52 -----
53 -----
54 CONTENIDO DE LA TABLA DE SIMBOLOS LOCAL DE LA FUNCION hornear #3 :
55
56 * LEXEMA : 'barraPan'
57 ATRIBUTOS:
58 + tipo: 'cadena'
59 + despl: 0
60 -----
61 -----

```

Figura 18: Fichero de TS *Panaderia.js*

- P (44)
 - B (29)
 - let
 - id
 - T (34)
 - string
 - ;
 - P (44)
 - B (29)
 - let
 - id
 - T (34)
 - string
 - ;
 - P (44)
 - B (29)
 - let
 - id
 - T (34)
 - string
 - ;
 - P (44)
 - B (29)
 - let
 - id
 - T (33)
 - boolean
 - ;
 - P (45)
 - F (35)
 - function
 - id
 - H (37)
 - lambda
 - (
 - A (38)
 - T (34)
 - string
 - id
 - K (40)
 - ;
 - T (34)
 - string
 - id
 - K (41)
 - lambda
 -)
 - {
 - C (42)
 - B (31)
 - do



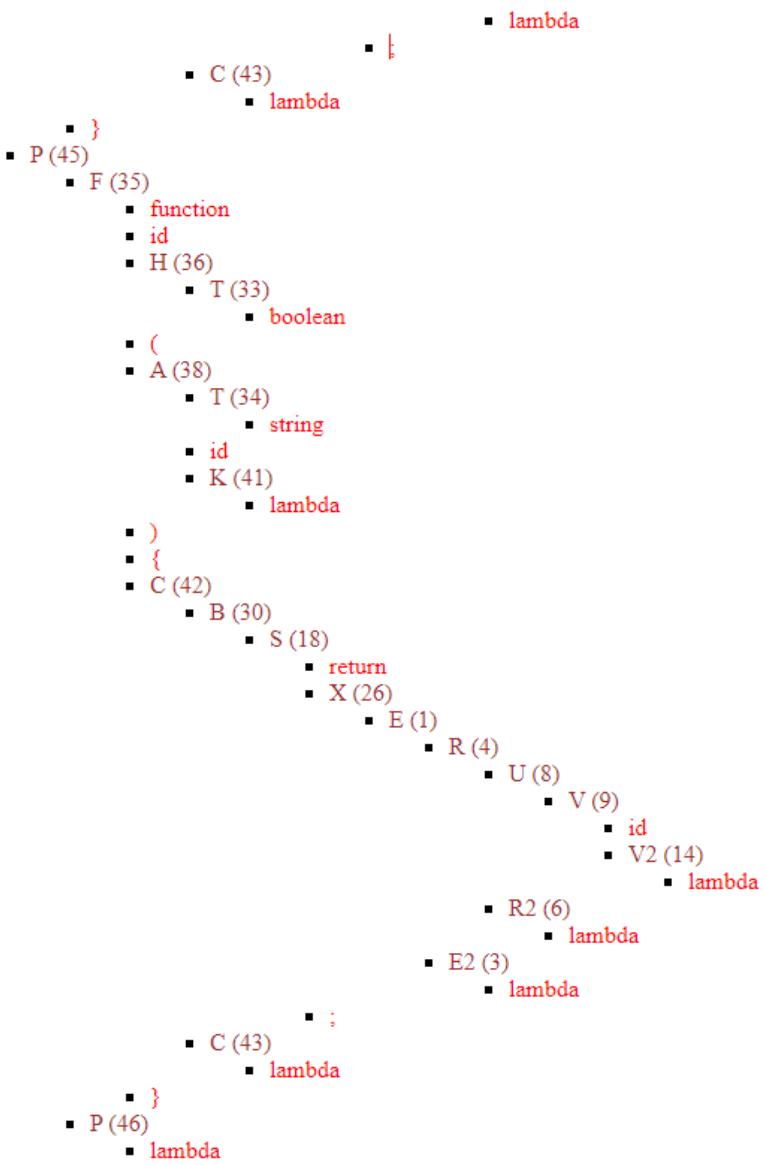


Figura 19: Fichero del árbol *Panaderia.js*

Caso 4

Entrada



```
1 let aguacate int;
2 let pomelo int;
3 let pina int;
4 let naranja int;
5 let rica boolean;
6 let ingredientes string;
7 ingredientes = 'aguacate, pomelo, pina, naranja';
8 let macedonia int;
9 macedonia = aguacate * pomelo * naranja * pina;
10
11 if( macedonia < 626)
12     rica = rica;
```

Figura 20: Fichero de entrada *Macedonia.js*

Salida



```
1 <1, > // token palabra reservada let
2 <25,0> // token identificador aguacate
3 <2, > // token palabra reservada int
4 <16, > // token puntoYcoma
5 <1, > // token palabra reservada let
6 <25,1> // token identificador pomelo
7 <2, > // token palabra reservada int
8 <16, > // token puntoYcoma
9 <1, > // token palabra reservada let
10 <25,2> // token identificador pina
11 <2, > // token palabra reservada int
12 <16, > // token puntoYcoma
13 <1, > // token palabra reservada let
14 <25,3> // token identificador naranja
15 <2, > // token palabra reservada int
16 <16, > // token puntoYcoma
17 <1, > // token palabra reservada let
18 <25,4> // token identificador rica
19 <4, > // token palabra reservada boolean
20 <16, > // token puntoYcoma
21 <1, > // token palabra reservada let
22 <25,5> // token identificador ingredientes
23 <3, > // token palabra reservada string
24 <16, > // token puntoYcoma
25 <25,5> // token identificador ingredientes
26 <18, > // token asign
27 <23,"aguacate, pomelo, pina, raranja"> // token cadena
28 <16, > // token puntoYcoma
29 <1, > // token palabra reservada let
30 <25,6> // token identificador macedonia
31 <2, > // token palabra reservada int
32 <16, > // token puntoYcoma
33 <25,6> // token identificador macedonia
34 <18, > // token asign
35 <25,0> // token identificador aguacate
36 <20, > // token mult
37 <25,1> // token identificador pomelo
38 <20, > // token mult
39 <25,3> // token identificador naranja
40 <20, > // token mult
41 <25,2> // token identificador pina
42 <16, > // token puntoYcoma
43 <5, > // token palabra reservada if
44 <12, > // token parIzq
45 <25,6> // token identificador macedonia
46 <22, > // token menor
47 <24,626> // token constante entera
48 <13, > // token parDrch
49 <25,4> // token identificador rica
50 <18, > // token asign
51 <25,4> // token identificador rica
52 <16, > // token puntoYcoma
```

Figura 21: Fichero de tokens *Macedonia.js*

```

● ● ●
1 Descendente 44 29 32 44 29 32 44 29 32 44 29 32 44 29 33 44 29 34 44 30 15 19 1 4 8
2 12 6 3 44 29 32 44 30 15 19 1 4 8 9 14 5 8 9 14 5 8 9 14 5 8 9 14 6 3 44 28 1 4 8 9
3 14 6 2 4 8 11 6 3 15 19 1 4 8 9 14 6 3 46

```

Figura 22: Fichero de parse *Macedonia.js*

```

● ● ●
1 CONTENIDO DE LA TABLA DE SIMBOLOS GLOBAL #1 :
2
3 * LEXEMA :      'aguacate'
4   ATRIBUTOS:
5   + tipo:      'entero'
6   + despl:     0
7 -----
8 * LEXEMA :      'pomelo'
9   ATRIBUTOS:
10  + tipo:      'entero'
11  + despl:     1
12 -----
13 * LEXEMA :      'pina'
14   ATRIBUTOS:
15   + tipo:      'entero'
16   + despl:     2
17 -----
18 * LEXEMA :      'naranja'
19   ATRIBUTOS:
20   + tipo:      'entero'
21   + despl:     3
22 -----
23 * LEXEMA :      'rica'
24   ATRIBUTOS:
25   + tipo:      'logico'
26   + despl:     4
27 -----
28 * LEXEMA :      'ingredientes'
29   ATRIBUTOS:
30   + tipo:      'cadena'
31   + despl:     5
32 -----
33 * LEXEMA :      'macedonia'
34   ATRIBUTOS:
35   + tipo:      'entero'
36   + despl:     69
37 -----
38 -----

```

Figura 23: Fichero de TS *Macedonia.js*

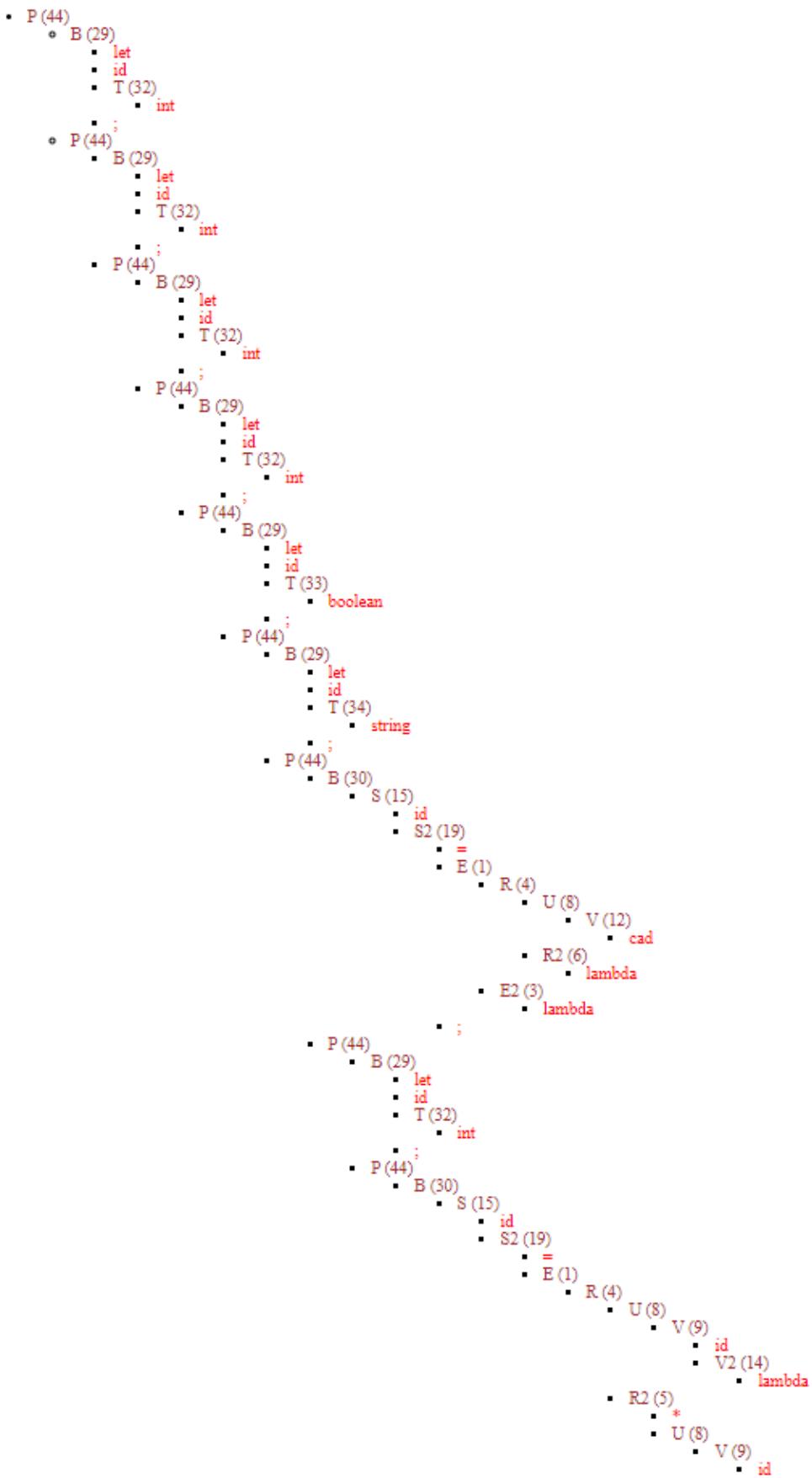




Figura 24: Fichero del árbol *Macedonia.js*

Caso 5

Entrada



```
1 let cadena string;
2 cadena= 'aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa';
3 let resu int;
4 let a boolean;
5 let i int;
6
7 i = 10;
8 function factorial ( int x){
9 do{
10     resu = resu*4444;
11 }
12 while(i < 0);
13 }
14 print(resu);
15
```

Figura 25: Fichero de entrada *Factorial.js*

Salida



```
1 <1, > // token palabra reservada let
2 <25,0> // token identificador cadena
3 <3, > // token palabra reservada string
4 <16, > // token puntoYcoma
5 <25,0> // token identificador cadena
6 <18, > // token asign
7 <23,"aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"> // token cadena
8 <16, > // token puntoYcoma
9 <1, > // token palabra reservada let
10 <25,1> // token identificador resu
11 <2, > // token palabra reservada int
12 <16, > // token puntoYcoma
13 <1, > // token palabra reservada let
14 <25,2> // token identificador a
15 <4, > // token palabra reservada boolean
16 <16, > // token puntoYcoma
17 <1, > // token palabra reservada let
18 <25,3> // token identificador i
19 <2, > // token palabra reservada int
20 <16, > // token puntoYcoma
21 <25,3> // token identificador i
22 <18, > // token asign
23 <24,10> // token constante entera
24 <16, > // token puntoYcoma
25 <8, > // token palabra reservada function
26 <25,4> // token identificador factorial
27 <12, > // token parIzq
28 <2, > // token palabra reservada int
29 <25,0> // token identificador x
30 <13, > // token parDrch
31 <14, > // token llaveIzq
32 <6, > // token palabra reservada do
33 <14, > // token llaveIzq
34 <25,1> // token identificador resu
35 <18, > // token asign
36 <25,1> // token identificador resu
37 <20, > // token mult
38 <24,4444> // token constante entera
39 <16, > // token puntoYcoma
40 <15, > // token llaveDrch
41 <7, > // token palabra reservada while
42 <12, > // token parIzq
43 <25,3> // token identificador i
44 <22, > // token menor
45 <24,0> // token constante entera
46 <13, > // token parDrch
47 <16, > // token puntoYcoma
48 <15, > // token llaveDrch
49 <10, > // token palabra reservada print
50 <12, > // token parIzq
51 <25,1> // token identificador resu
52 <13, > // token parDrch
53 <16, > // token puntoYcoma
```

Figura 26: Fichero de tokens *Factorial.js*

```
1 Descendente 44 29 34 44 30 15 19 1 4 8 12 6 3 44 29 32 44 29 33 44 29 32
2 44 30 15 19 1 4 8 11 6 3 45 35 37 38 32 41 42 31 42 30 15 19 1 4 8 9 14
3 5 8 11 6 3 43 1 4 8 9 14 6 2 4 8 11 6 3 43 44 30 16 1 4 8 10 1 4 8 9 14
4 6 3 6 3 46
```

Figura 27: Fichero de parse *Factorial.js*

```
1 CONTENIDO DE LA TABLA DE SIMBOLOS GLOBAL #1 :
2
3 * LEXEMA : 'cadena'
4 ATRIBUTOS:
5 + tipo: 'cadena'
6 + despl: 0
7 -----
8 * LEXEMA : 'resu'
9 ATRIBUTOS:
10 + tipo: 'entero'
11 + despl: 64
12 -----
13 * LEXEMA : 'a'
14 ATRIBUTOS:
15 + tipo: 'logico'
16 + despl: 65
17 -----
18 * LEXEMA : 'i'
19 ATRIBUTOS:
20 + tipo: 'entero'
21 + despl: 66
22 -----
23 * LEXEMA : 'factorial'
24 ATRIBUTOS:
25 + tipo: 'funcion'
26 + numParam: 1
27 + TipoParam1: 'entero'
28 + TipoRetorno: 'vacio'
29 + EtiqFuncion: 'EtFun1'
30 -----
31 -----
32 CONTENIDO DE LA TABLA DE SIMBOLOS LOCAL DE LA FUNCION factorial #2 :
33
34 * LEXEMA : 'x'
35 ATRIBUTOS:
36 + tipo: 'entero'
37 + despl: 0
38 -----
39 -----
```

Figura 28: Fichero de TS *Factorial.js*

- P (44)
 - B (29)
 - let
 - id
 - T (34)
 - string
 - ;
 - P (44)
 - B (30)
 - S (15)
 - id
 - S2 (19)
 - =
 - E (1)
 - R (4)
 - U (8)
 - V (12)
 - cad
 - R2 (6)
 - lambda
 - E2 (3)
 - lambda
 - ;
 - P (44)
 - B (29)
 - let
 - id
 - T (32)
 - int
 - ;
 - P (44)
 - B (29)
 - let
 - id
 - T (33)
 - boolean
 - ;
 - P (44)
 - B (29)
 - let
 - id
 - T (32)
 - int
 - ;
 - P (44)
 - B (30)
 - S (15)
 - id
 - S2 (19)
 - =
 - E (1)
 - R (4)
 - U (8)
 - V (11)
 - ent
 - R2 (6)
 - lambda
 - E2 (3)
 - lambda
 - ;
 - P (45)

```

F (35)
  - function
  - id
  - H (37)
    - lambda
  - (
  - A (38)
    - T (32)
      - int
    - id
    - K (41)
      - lambda
  - )
  - {
  - C (42)
    - B (31)
      - do
    - {
    - C (42)
      - B (30)
        - S (15)
          - id
        - S2 (19)
          - =
        - E (1)
          - R (4)
            - U (8)
              - V (9)
                - id
                - V2 (14)
                  - lambda
            - R2 (5)
              - *
            - U (8)
              - V (11)
                - ent
            - R2 (6)
              - lambda
            - E2 (3)
              - lambda
  - ;
  - C (43)
    - lambda
  - }
  - while
  - (
  - E (1)
    - R (4)
      - U (8)
        - V (9)
          - id
          - V2 (14)
            - lambda

```

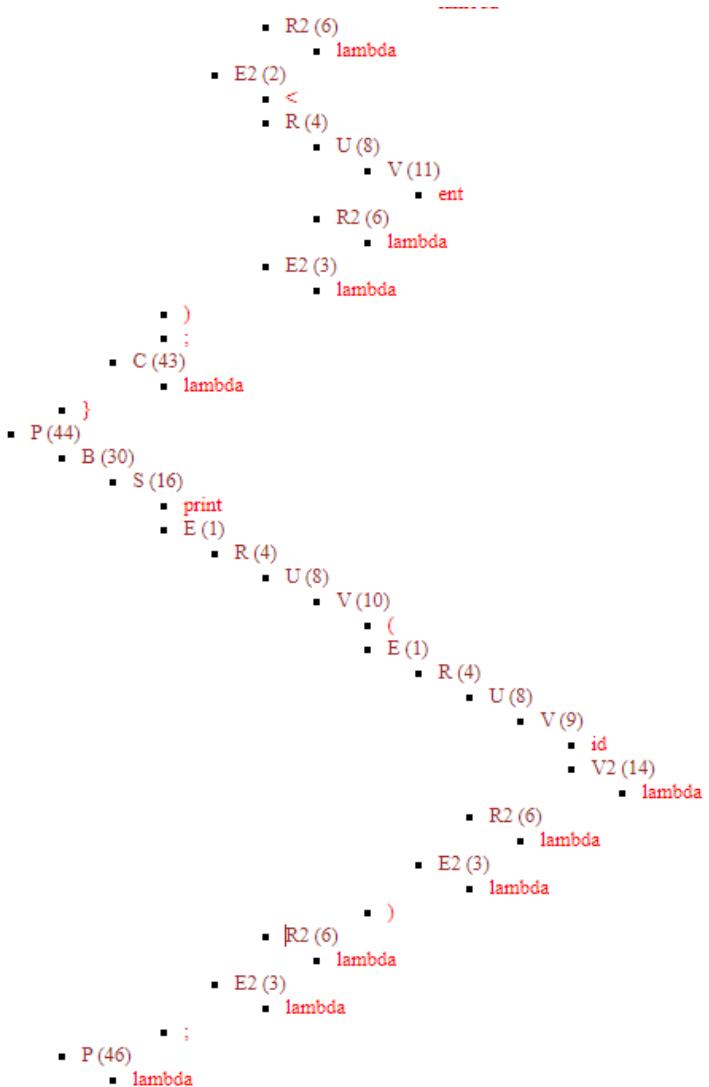


Figura 29: Fichero del árbol *Factorial.js*

Casos con errores

En segundo lugar, se proponen una serie de casos prueba en los que figuran errores semánticos. Por tanto, se debería una traza descriptiva del error encontrado en el fichero de volcado. Se harán los casos en un orden creciente de dificultad.

Caso 1

Entrada



```
1 let a int;
2 a = 3;
3 b = a;
4 let string c;
5
6 a %= b;
```

Figura 30: Fichero de entrada *Modulo.js*

El procesador debería encontrar un error por primera vez en la línea 4, ya que se hace la declaración de variables en un orden incorrecto.

Salida

El procesador, en efecto, devuelve el siguiente error en el fichero de errores:



```
1 Error semántico (29): Línea 4: El orden para declarar una variable es: let + identificador + tipo
2 Error sintáctico (6): Línea 4: Se ha encontrado 'string' y se esperaba 'identificador (nombre de una variable o función)'
```

Figura 31: Fichero de errores *Modulo.js*

Caso 2

Entrada



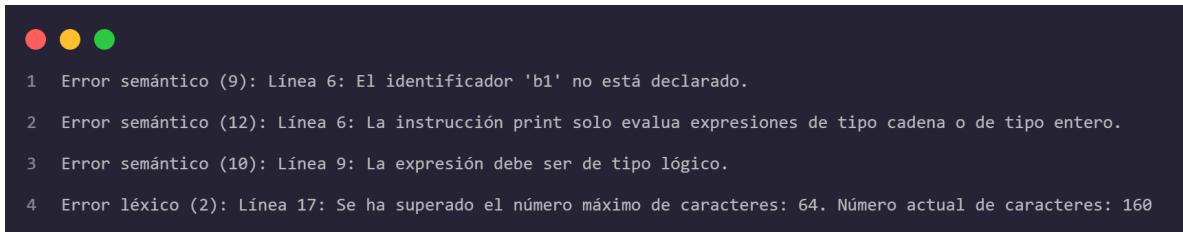
```
1 let msg string;           // declaracion de variables
2 let recibido boolean;
3
4 i = 0;
5 do{
6     print b1;
7     i += 1;
8 }
9 while(msg);
10
11 function pideTexto ()      // funcion
12 {
13     print 'Introduce una palabra.....';
14     .....
15     .....
16     .....
17     .....';
18     input msg;
19 }
20
21 pideTexto();
```

Figura 32: Fichero de entrada *MsgLargo.js*

El analizador debería encontrar un error por primera vez en la línea 6, ya que el identificador *b1* no esta declarado, y el último debe encontrarlo en la línea 17, ya que se emplean 160 caracteres para la cadena y se supera el máximo permitido (64).

Salida

El procesador, en efecto, devuelve el siguiente error en el fichero de errores:



```
1 Error semántico (9): Línea 6: El identificador 'b1' no está declarado.
2 Error semántico (12): Línea 6: La instrucción print solo evalua expresiones de tipo cadena o de tipo entero.
3 Error semántico (10): Línea 9: La expresión debe ser de tipo lógico.
4 Error léxico (2): Línea 17: Se ha superado el número máximo de caracteres: 64. Número actual de caracteres: 160
```

Figura 33: Fichero de errores *MsgLargo.js*

Caso 3

Entrada

```
1 let a int;
2 let b int;
3 let number string;
4 let log boolean;
5
6 function operacion
7     int (int num1_, int num2_)
8 {
9     number = 110;
10    let number int;
11    number = 88 * num1_*num2_;
12    let bool boolean ;
13    bool = number < 10;
14    if (bool) print('1101110');
15    return bool;
16 }
17 number += operacion (a, b);
18 print !number;
19 avocado = 'peyala';
```

Figura 34: Fichero de entrada *Operacion.js*

El analizador debería encontrar un error por primera vez en la línea 9, ya que la variable global *number* es de tipo cadena y se le intenta asignar un número entero. Sin embargo, en la línea 11 no debería tener este problema, puesto que en la 10 se declara una variable del mismo nombre de tipo entero. Por tanto, la función debería emplear la local. Entre otros errores intermedios, debería encontrar el último en la línea 19, puesto que la variable *avocado* se declara implícitamente y por tanto es de tipo entera. Al intentar asignarle una cadena, debería dar un error.

Salida

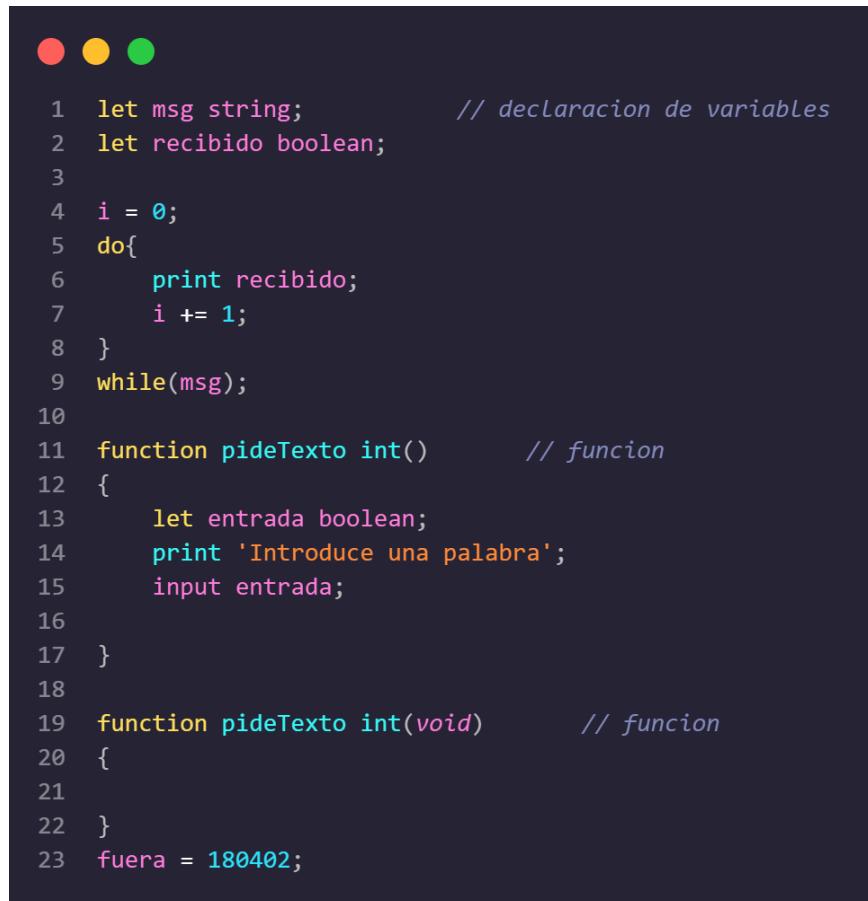
El procesador, en efecto, devuelve el siguiente error en el fichero de errores:

```
1 Error semántico (23): Línea 9: Ambos lados de la expresión deben de ser de tipo cadena.
2 Error semántico (20): Línea 15: El tipo de retorno (logico) no coincide con el definido en la función (entero).
3 Error semántico (24): Línea 17: Los operadores aritméticos (+=) solo se aplican sobre tipos enteros.
4 Error semántico (26): Línea 18: Los operadores lógicos (!) solo se aplican sobre tipos lógicos.
5 Error semántico (12): Línea 18: La instrucción print solo evalúa expresiones de tipo cadena o de tipo entero.
6 Error semántico (23): Línea 19: Ambos lados de la expresión deben de ser de tipo entero.
```

Figura 35: Fichero de errores *Operacion.js*

Caso 4

Entrada



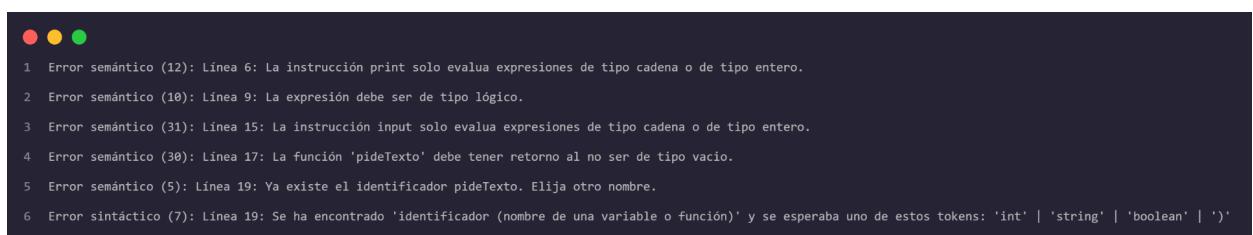
```
1 let msg string;           // declaracion de variables
2 let recibido boolean;
3
4 i = 0;
5 do{
6     print recibido;
7     i += 1;
8 }
9 while(msg);
10
11 function pideTexto int()      // funcion
12 {
13     let entrada boolean;
14     print 'Introduce una palabra';
15     input entrada;
16
17 }
18
19 function pideTexto int(void)    // funcion
20 {
21
22 }
23 fuera = 180402;
```

Figura 36: Fichero de entrada *Mensajeria.js*

El analizador debería encontrar un error por primera vez en la línea 6, ya que la instrucción `print` solo evalúa expresiones de tipo entero o cadena, y la variable `recibido` es de tipo lógico. Entre otros errores intermedios, debería encontrar el último en la línea 19, puesto que se produce un error léxico al introducir un token no esperado en la definición de la función. De esta manera, no debería mostrarse el error léxico de la línea 23, donde el número se sale del rango representable de constantes enteras.

Salida

El procesador, en efecto, devuelve el siguiente error en el fichero de errores:



```
1 Error semántico (12): Línea 6: La instrucción print solo evalúa expresiones de tipo cadena o de tipo entero.
2 Error semántico (10): Línea 9: La expresión debe ser de tipo lógico.
3 Error semántico (31): Línea 15: La instrucción input solo evalúa expresiones de tipo cadena o de tipo entero.
4 Error semántico (30): Línea 17: La función 'pideTexto' debe tener retorno al no ser de tipo vacío.
5 Error semántico (5): Línea 19: Ya existe el identificador pideTexto. Elija otro nombre.
6 Error sintáctico (7): Línea 19: Se ha encontrado 'identificador (nombre de una variable o función)' y se esperaba uno de estos tokens: 'int' | 'string' | 'boolean' | ''
```

Figura 37: Fichero de errores *Mensajeria.js*

Caso 5

Entrada



```
1 function esFechaCorrecta boolean (int d, int m, int a)
2 {
3     return esFechaCorrecta(1,esFechaCorrecta(1,1,0),1);
4 }
```

Figura 38: Fichero de entrada *Fecha.js*

El analizador debería encontrar un error por única vez en la línea 3, ya que se devuelve una llamada recursiva de la función en cuyos parámetros se vuelve a llamar recursivamente a la función. Como la función retorna un tipo lógico y el segundo parámetro de la función requiere un tipo entero, debería identificarlo correctamente.

Salida

El procesador, en efecto, devuelve el siguiente error en el fichero de errores:



```
1 Error semántico (11): Línea 3: Se ha encontrado un parámetro de tipo logico y se esperaba un parámetro de tipo entero.
```

Figura 39: Fichero de errores *Fecha.js*

Última modificación: 11/01/2023 a las 13:24