Práctica 2 - Procesadores de Lenguajes

Grupo 7
Carmen Toribio Pérez, 22M009
Sergio Gil Atienza, 22M046
María Moronta Carrión, 22M111

Introducción

La primera entrega de esta práctica consistió en la implementación de un **Analizador Léxico** y una **Tabla de Símbolos**. Para ello, seguimos los siguientes pasos:

1. **Identificar los Tokens** del lenguaje fuente, teniendo en cuenta las especificaciones de nuestro grupo.
2. Construir la **Gramática Regular** que los genera.
3. Diseñar el **Autómata Finito Determinista** equivalente a la gramática. Hemos realizado una representación de este a través de un **diagrama de estados**.
4. Añadir las **Acciones Semánticas**, asociadas a cada una de las transiciones del diagrama de estados.
5. Estudiar los posibles **casos de error**, para poder manejar correctamente la detección y el reporte de errores léxicos.
6. Comienzo de la implementación de la **Tabla de Símbolos**, concretamente con el diseño de su estructura y su organización.
7. Demostración del **funcionamiento del programa del Analizador Léxico**, gracias a la presentación de una serie de casos de prueba.

Esta segunda entrega consiste en la implementación de un **Analizador Sintáctico**, para lo cual hemos seguido los siguientes pasos:

1. **C**onstruir la **Gramática de Contexto Libre** basándonos en las reglas del lenguaje fuente.
2. Demostración de que se trata de una **Gramática LL(1)**
3. Diseño en pseudo-código de las **funciones del Analizador** (una por cada símbolo no terminal de la gramática).
4. Demostración del **funcionamiento del programa del Analizador Sintáctico**, gracias a la presentación de una serie de casos de prueba.

Como integrantes del **grupo 7**, hemos tenido que cumplir con las siguientes especificaciones:

- Sentencias: Sentencia repetitiva (for)
- Operadores especiales: Asignación con suma (+=)
- Técnicas de Análisis Sintáctico: Descendente Recursivo
- Comentarios: Comentario de bloque (/* */)
- Cadenas: Con comillas simples (' ')

1 Tokens

El primer paso a la hora de construir un Analizador Léxico es la identificación de los tokens. Para hacer la lista nos hemos basado en la actividad práctica de la plataforma Draco. Hemos decidido utilizar el mismo formato en tablas con tal de facilitar su legibilidad.
Tokens obligatorios:

Elemento	Código de Token	Atributo
boolean	bool	-
for	for	-
function	fn	-
if	if	-
input	in	-
int	int	-
output	out	-
return	ret	-
string	str	-
var	var	-
void	void	-
constante entera	cint	Número
Cadena (')	cstr	Cadena ("c*")
Identificador	id	Número (posición en la TS)
+=	cumass	-
=	ass	-
,	com	-
;	scol	-
(	po	-
)	pc	-
{	cbo	-
}	cbc	-

Tokens de operadores aritméticos, lógicos y relacionales:

Grupo de Opciones	Código de Token	Atributo
Grupo Operadores Aritméticos: Suma (+)	sum	-
Grupo Operadores Aritméticos: Resta (-)	sub	-
Grupo Operadores Lógicos: Y lógico (&&)	and	-
Grupo Operadores Lógicos: O lógico (||)	or	-
Grupo Operadores Relacionales: Menor (<)	ls	-
Grupo Operadores Relacionales: Mayor (>)	gr	-

Tokens opcionales:

Grupo de Opciones	Código de Token	Atributo
false	cap	-
true	nocap	-
EOF	eof	-

Por tanto, los siguientes tipos de expresiones no serán identificados como tokens: los delimitadores (como los espacios en blanco o las tabulaciones), los comentarios de bloque (/* */) o los saltos de línea (\n).

2 Gramática Regular del Analizador Léxico

En esta sección, describimos la Gramática Regular (gramática de tipo 3 según la jerarquía de Chomsky) que hemos diseñado para identificar y generar los tokens del lenguaje fuente.

Símbolos no terminales

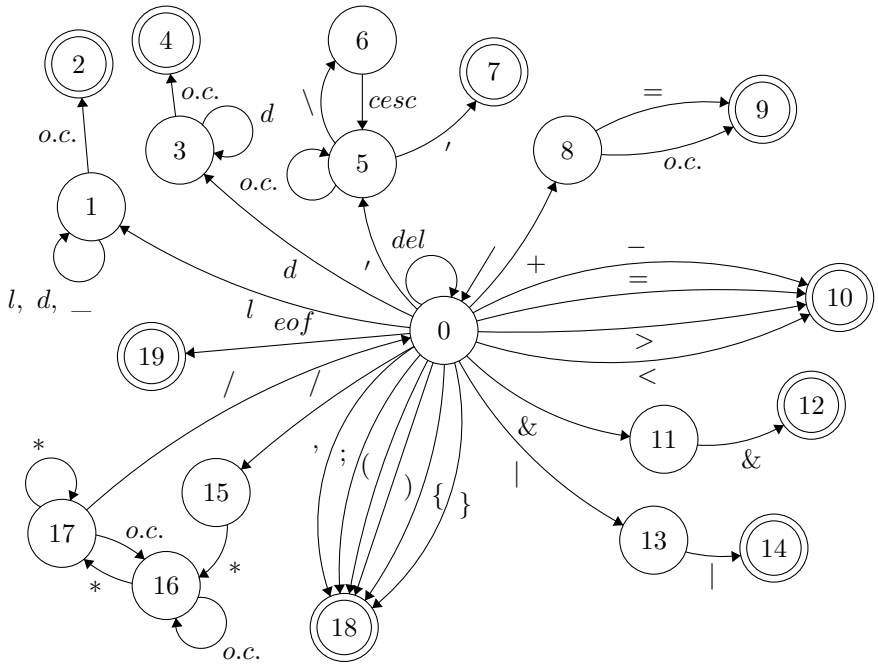
d := 0...9
l := a...z, A...Z
c_1 := espacio o cualquier carácter imprimible menos \
$cesc$:= carácter escapable (', 0, n, a, t, v, f, r, \)
c_2 := cualquier carácter menos * y eof
c_3 := cualquier carácter menos *, / y eof

Gramática Regular

S → del S | lA | dB | 'C | +D| - | = | > | < | &E | |F | /G | } | { |) | (| ; | , | eof
A → lA | dA | _A | λ
B → dB | λ
C → c_1C | \C' | '
C' → $cesc$C
D → = | λ
E → &
F → |
G → *H
H → c_2H | *I
I → /S | c_3H | *I

3 Autómata Finito Determinista

Una vez definida la gramática regular, el siguiente paso es construir el Autómata Finito Determinista (AFD) correspondiente. A continuación, presentamos su diseño, incluyendo las transiciones entre estados.



4 Acciones semánticas

A lo largo de esta sección, describimos las acciones semánticas que hemos añadido a las transiciones del Autómata Finito Determinista. Estas acciones permiten que se lleven a cabo acciones como la lectura, la identificación de los tokens o, en su lugar, una correcta gestión de los errores léxicos. A continuación, detallamos las operaciones utilizadas y, por legibilidad en el AFD, las transiciones en las que se realiza cada una:

Leer:

```
car := leer();          // Todas las transiciones salvo 1:2, 3:4, 6:5, 8:9 con un carácter
                        // distinto de '=', y 0:19
car := leerCesc();      // Transición 6:5. Leído un caracter de escape, devuelve el carácter
                        // correspondiente (por ejemplo un 'n' devuelve un eol)
```

Concatenar:

```
lex := car;             // Transición 0:1
lex := ∅;               // Transición 0:5
lex := lex ⊕ car;       // Transiciones 1:1, 5:5 y 6:5
```

Contador:

```
// Utilizado para contar el número de caracteres de la cadena
cont := 0;              // Transición 0:5. Inicializa el contador a 0
cont := cont + 1;       // Transiciones 5:5 y 6:5
```

Calcular valor entero:

```
// La función val(car) devuelve el valor entero del dígito correspondiente al carácter car
num := val(car);            // Transición 0:3
num := num * 10 + val(car); // Transición 3:3
```

Generar token:

Cadenas y enteros

```
G1  if (cont > 64)                              // Transición 5:7
        then error (COD_ERROR_STRLEN, lex)
    else Gen_token(cstr, lex)
G2  if (num > 32767)                            // Transición 3:4
        then error (COD_ERROR_MAXINT, num)
    else Gen_token(cint, num)
```

Delimitadores y operadores de control

```
G3  Gen_token(cumass, -)        // Transición 8:9 con el carácter '='
G4  Gen_token(ass, -)           // Transición 0:10 con el carácter '='
G5  Gen_token(com, -)           // Transición 0:18 con el carácter ','
G6  Gen_token(scol, -)          // Transición 0:18 con el carácter ';'
G7  Gen_token(po, -)            // Transición 0:18 con el carácter '('
G8  Gen_token(pc, -)            // Transición 0:18 con el carácter ')'
G9  Gen_token(cbo, -)           // Transición 0:18 con el carácter '{'
G10 Gen_token(cbc, -)           // Transición 0:18 con el carácter '}'
```

Operadores Aritméticos, Lógicos y Relacionales

```
G11 Gen_token(sum, -)           // Transición 8:9 con un carácter distinto de '='
G12 Gen_token(sub, -)           // Transición 0:10 con el carácter '-'
G13 Gen_token(and, -)           // Transición 11:12
G14 Gen_token(or, -)            // Transición 13:14
G15 Gen_token(ls, -)            // Transición 0:10 con el carácter '<'
G16 Gen_token(gr, -)            // Transición 0:10 con el carácter '>'
```

Fin de fichero (EOF)

```
G17 Gen_token(eof, -)           // Transición 0:19
```

Identificadores y palabras reservadas

```
G18   type := GetTokenCode(lex);*                    // Transición 1:2
      if type = id then
            pos := SearchST(lex);**
            if pos = NULL then
                  pos := AddID(lex);***
            Gen_Token(type, pos);
      else
            Gen_Token(type, -);
```

* La función GetTokenCode(lex) devuelve el código de token de la palabra reservada con la que coincida lex, o el código de ID si no es palabra reservada

** La función SearchST(lex) devuelve la posición del identificador lex en la tabla de símbolos, o NULL si no ha sido insertado aún

*** La función AddID(lex) inserta el identificador lex en la tabla de símbolos y devuelve su posición

5 Gestión de errores

Los casos de error posibles son todas aquellas transiciones no recogidas por nuestro AFD. En estos casos, se generará un error léxico y se informará al usuario de la existencia de un error en el código fuente, junto con el número de línea y columna en la que se ha detectado el mismo. A continuación, detallamos los errores que hemos identificado y los mensajes de error correspondientes a cada uno:

- **Caracteres no reconocidos.** En caso de que el carácter leído no coincida con ninguna transición del AFD, de manera que no pueda transitar desde el estado 0, se generará un error léxico con el siguiente mensaje: (línea:columna) ERROR: Carácter inesperado al buscar el siguiente símbolo («carácter inesperado», U+:04X).
- **Comentario de bloque mal abierto.** En caso de que tras encontrar una '/' no se encuentre un «*», se generará un error léxico con el siguiente mensaje: (línea:columna) ERROR: Carácter inesperado tras «/». Se esperaba «*» para abrir un comentario de bloque
- **Comentario de bloque no cerrado.** Si se detecta un comentario de bloque sin cerrar, se generará un error léxico con el siguiente mensaje: (línea:columna) ERROR: Fin de fichero inesperado. Se esperaba «*/» para cerrar el comentario de bloque.
- **Valor entero fuera de rango.** Si el valor de una constante entera supera el rango permitido que abarcan los enteros de 16 bits con signo, se generará un error léxico con el siguiente mensaje: (línea:columna) ERROR: El valor del entero es demasiado grande (máximo 32767).
- **Cadena de caracteres no cerrada.** Si se detecta una cadena de caracteres sin cerrar, se generará un error léxico con el siguiente mensaje: (línea:columna) ERROR: Fin de fichero inesperado. Se esperaba «'» para cerrar la cadena.
- **Secuencia de escape no válida.** En este caso, distinguiremos dos situaciones:
  - **Carácter imprimible tras la barra invertida.** Si el carácter siguiente a la barra invertida es un carácter imprimible, pero no forma ninguna secuencia de escape válida, se generará un error léxico con el siguiente mensaje: (línea:columna) ERROR: Error en la cadena, la secuencia de escape «\carácter» (U+:04X) no es válida.
  - **Carácter ilegal tras la barra invertida.** Si el carácter que sigue a la barra invertida no es un carácter imprimible, se generará un error léxico con el siguiente mensaje: (línea:columna) ERROR: Error en la cadena, carácter ilegal en la secuencia de escape (U+:04X).

- **Carácter no permitido en la cadena.** Si se detecta un carácter no permitido en la cadena, como un salto de línea, se generará un error léxico con el siguiente mensaje: (línea:columna) ERROR: Error en la cadena, carácter no permitido (U+:04X).
- **Cadena de caracteres demasiado larga.** Si la longitud de la cadena supera los 64 caracteres, se generará un error léxico con el siguiente mensaje (donde x es el número de caracteres de la cadena): (línea:columna) ERROR: La longitud de cadena excede el límite de 64 caracteres (x caracteres).
- **Operadores lógicos incorrectos.** En caso de haber un `&` en lugar de `&&` o un `|` en lugar de `||`, se generará un error léxico con el siguiente mensaje (donde op será `&` o `|`): (línea:columna) ERROR: Se esperaba «op» después de «op» para formar un operador.

Para ilustrar mejor el comportamiento del programa, en la sección 8 presentamos varios casos en los que demostramos cómo el analizador maneja la detección de errores.

6 Tabla de Símbolos

Dado que en esta parte de la práctica aún no hemos implementado el **analizador semántico**, nuestra tabla de símbolos es única y global. Por el momento, simplemente almacena los identificadores que el analizador léxico encuentre, y carecerá de atributos más allá del nombre del propio identificador.

7 Funcionamiento del programa

El programa que hemos elaborado comienza leyendo el código fuente que le proporcionamos. Lo analiza carácter por carácter, reconociendo los tokens que hemos definido en el punto 2. Para ello, el Analizador Léxico se basa en la gramática regular que genera nuestro Autómata Finito Determinista.

A medida que encuentra identificadores los almacena en la tabla de símbolos si es necesario. Si encuentra delimitadores o comentarios de bloque, los ignora. Además, el programa está preparado para detectar errores, generando mensajes claros y concisos que indican el tipo de error y la ubicación exacta, con tal de facilitar la corrección por parte del usuario. Al finalizar la ejecución, queda completo un informe con todos los tokens que el programa ha reconocido durante el análisis, y los posibles errores que haya encontrado.

En el **anexo** es posible encontrar múltiples ejemplos que demuestran el funcionamiento correcto del programa. Los tres primeros son favorables y los tres últimos muestran diversos casos de error. A continuación incluimos una breve explicación de cada uno:

- **Caso 1:** El programa reconoce declaraciones válidas, como la asignación de tipos a variables y la asignación de identificadores, como lo que ocurre con el tipo «bool» (tratado como un identificador).
- **Caso 2:** Muestra el funcionamiento adecuado de una función println (para imprimir mensajes), con lo que se puede apreciar la correcta gestión de cadenas y el uso de saltos de línea.
- **Caso 3:** Demuestra la capacidad del programa para comparar variables de entrada y ejecutar operaciones matemáticas.
- **Caso 4:** Ilustra un error léxico causado por un comentario de bloque no cerrado, que resulta en un fin de fichero inesperado.
- **Caso 5:** Muestra que hay caracteres no permitidos al inicio de un token y que se produce un error si declaramos variables que no cumplen con las convenciones, como empezar por «_».
- **Caso 6:** Resalta cómo el programa continúa buscando tokens válidos a pesar de haber errores en el código, lo que permite depurar varios errores en una sola ejecución.

Con esto, demostramos el funcionamiento del programa en diferentes situaciones, desde el reconocimiento de declaraciones válidas hasta la gestión de errores léxicos. Estos resultados serán completados gracias a la segunda entrega del proyecto, que se centrará en el desarrollo del Analizador Sintáctico.

8 Gramática de Contexto Libre del Analizador Sintáctico

En esta sección, describimos la Gramática de Contexto Libre (gramática de tipo 2 según la jerarquía de Chomsky) que hemos diseñado para representar la estructura sintáctica del lenguaje fuente.

1. P → FUNCTION P
2. P → STATEMENT P
3. P → *eof*
4. FUNCTION → function FUNTYPE *id* (FUNATTRIBUTES) { BODY }
5. FUNTYPE → *void*
6. FUNTYPE → VARTYPE
7. VARTYPE → *int*
8. VARTYPE → *boolean*
9. VARTYPE → *string*
10. FUNATTRIBUTES → *void*
11. FUNATTRIBUTES → VARTYPE *id* NEXTATTRIBUTE
12. NEXTATTRIBUTE → , VARTYPE *id* NEXTATTRIBUTE
13. NEXTATTRIBUTE → λ
14. BODY → STATEMENT BODY
15. BODY → λ
16. STATEMENT → *if* (EXP) ATOMSTATEMENT
17. STATEMENT → *for* (FORACT ; EXP ; FORACT) { BODY }
18. STATEMENT → *var* VARTYPE *id* ;
19. STATEMENT → ATOMSTATEMENT
20. ATOMSTATEMENT → *id* IDACT ;
21. ATOMSTATEMENT → *output* EXP ;
22. ATOMSTATEMENT → *input id* ;
23. ATOMSTATEMENT → *return* RETURNEXP ;
24. IDACT → ASS EXP
25. IDACT → (CALLPARAM)
26. FORACT → *id* ASS EXP
27. FORACT → λ
28. ASS → =
29. ASS → +=
30. CALLPARAM → EXP NEXTPARAM
31. CALLPARAM → λ
32. NEXTPARAM → , EXP NEXTPARAM
33. NEXTPARAM → λ
34. RETURNEXP → EXP
35. RETURNEXP → λ
36. EXP → A EXP1
37. EXP1 → LOGOP A EXP1
38. EXP1 → λ
39. LOGOP → &&
40. LOGOP → ||
41. A → B A1
42. A1 → COMPOP B A1
43. A1 → λ
44. COMPOP → >
45. COMPOP → <
46. B → EXPATOM B1
47. B1 → ARITHMETICOP EXPATOM B1
48. B1 → λ
49. ARITHMETICOP → +
50. ARITHMETICOP → -
51. EXPATOM → *id* C
52. EXPATOM → (EXP)
53. EXPATOM → *cint*
54. EXPATOM → *cstr*
55. EXPATOM → *true*
56. EXPATOM → *false*
57. C → (CALLPARAM)
58. C → λ

9 Comprobación de la condición LL(1)

Por una parte, la gramática diseñada **no es ambigua**, ya que no existen dos producciones distintas para un mismo no terminal que generen la misma cadena de terminales y no terminales.

Además, **no es recursiva por la izquierda**, pues ninguna de las producciones de la misma es del tipo $A \rightarrow A\alpha$ (donde A es un no terminal y α es una cadena de terminales y no terminales).

No obstante, dado que múltiples reglas de la gramática diseñada tienen dos o más producciones distintas, es necesario realizar la **comprobación de la condición LL(1)**:

$$\forall\ A \in N, \text{ para cada par de reglas } A \rightarrow \alpha \mid \beta$$
$$\text{- Se cumple que } FIRST(\alpha) \cap FIRST(\beta) = \emptyset.$$
$$\text{- Si } \beta \text{ puede derivar } \lambda, \text{ entonces } FIRST(\alpha) \cap FOLLOW(A) = \emptyset.$$

De esta forma, podemos observar claramente que la gramática diseñada es LL(1):

P → FUNCTION P | STATEMENT P | *eof*
- FIRST(FUNCTION P) = {function}
- FIRST(STATEMENT P) = {if, for, var, id, output, input, return}
- FIRST(*eof*) = {eof}
  - FIRST(FUNCTION P) ∩ FIRST(STATEMENT P) = ∅
  - FIRST(FUNCTION P) ∩ FIRST(*eof*) = ∅
  - FIRST(STATEMENT P) ∩ FIRST(*eof*) = ∅

FUNTYPE → *void* | VARTYPE
- FIRST(*void*) = {void}
- FIRST(VARTYPE) = {int, boolean, string}
  - FIRST(*void*) ∩ FIRST(VARTYPE) = ∅

VARTYPE → *int* | *boolean* | *string*
- FIRST(*int*) = {int} ; FIRST(*boolean*) = {boolean} ; FIRST(*string*) = {string}
  - FIRST(*int*) ∩ FIRST(*boolean*) = ∅
  - FIRST(*int*) ∩ FIRST(*string*) = ∅
  - FIRST(*boolean*) ∩ FIRST(*string*) = ∅

FUNATTRIBUTES → *void* | VARTYPE *id* NEXTATTRIBUTE
- FIRST(*void*) = {void}
- FIRST(VARTYPE *id* NEXTATTRIBUTE) = {int, boolean, string}
  - FIRST(*void*) ∩ FIRST(VARTYPE *id* NEXTATTRIBUTE) = ∅

NEXTATTRIBUTE → , VARTYPE *id* NEXTATTRIBUTE | λ
- FIRST(, VARTYPE *id* NEXTATTRIBUTE) = {,}
- FIRST(λ) = {λ}
- FOLLOW(NEXTATTRIBUTE) = {)}
  - FIRST(, VARTYPE *id* NEXTATTRIBUTE) ∩ FIRST(λ) = ∅
  - FIRST(, VARTYPE *id* NEXTATTRIBUTE) ∩ FOLLOW(NEXTATTRIBUTE) = ∅

Procederíamos de esta forma con el resto de reglas de la gramática con dos o más producciones asociadas, pero dado que el proceso es tedioso y repetitivo, hemos decidido no incluirlo en la memoria. Cabe destacar, además, que la comprobación de la condición LL(1) ha sido contrastada exitosamente con los resultados obtenidos por la herramienta de apoyo ofrecida en la página web del departamento.

Anexo

1. **Caso 1:** Funcionamiento correcto

Código fuente

```
/* Declaraciones válidas */
var boolean a;
var int b;
var string d;

/* Se vuelve a declarar la variable «a», pero no es error léxico. */
var int a;

/* El tipo «bool» no existe, se trata como identificador. */
var bool err;
```

Volcado del fichero de tokens

```
<var, >
<bool, >
<id, 0>
<scol, >
<var, >
<int, >
<id, 1>
<scol, >
<var, >
<str, >
<id, 2>
<scol, >
<var, >
<int, >
<id, 0>
<scol, >
<var, >
<id, 3>
<id, 4>
<scol, >
<eof, >
```

Volcado del fichero de la tabla de símbolos

```
Tabla Global #0:
*'a'
*'b'
*'d'
*'bool'
*'err'
```

2. **Caso 2:** Funcionamiento correcto

Código fuente

```
function void println(string s) {
    output s;
    output '\n';
}

println('¡Hola mundo!');
println('Eso son llamadas a \'output\' usando una función.');
```

Volcado del fichero de tokens

```
<fn, >
<void, >
<id, 0>
<po, >
<str, >
<id, 1>
<pc, >
<cbo, >
<out, >
<id, 1>
<scol, >
<out, >
<cstr, "\n">
<scol, >
<cbc, >
<id, 0>
<po, >
<cstr, "¡Hola mundo!">
<pc, >
<scol, >
<id, 0>
<po, >
<cstr, "Eso son llamadas a \'output\' usando una función.">
<pc, >
<scol, >
<eof, >
```

Volcado del fichero de la tabla de símbolos

```
Tabla Global #0:
*'println'
*'s'
```

3. **Caso 3:** Funcionamiento correcto

Código fuente

```
/* Leemos dos números del usuario. Las variables sin declarar se suponen globales y enteras. */
input a;
input b;

/* Comparamos los números entre sí. */
if (a < b) {
    output '\'a\' es menor que \'b\'.';
}
if (a > b) {
    output '\'a\' es mayor que \'b\'.';
}

output '\n';

/* Operamos con los números. */

output 'a + b: ';
output a + b;

output 'a - b: ';
output a - b;

output '\n';
```

Volcado del fichero de tokens

```
<in, >
<id, 0>
<scol, >
<in, >
<id, 1>
<scol, >
<if, >
<po, >
<id, 0>
<ls, >
<id, 1>
<pc, >
<cbo, >
<out, >
<cstr, "\'a\' es menor que \'b\'.">
<scol, >
<cbc, >
<if, >
<po, >
<id, 0>
<gr, >
<id, 1>
<pc, >
<cbo, >
<out, >
<cstr, "\'a\' es mayor que \'b\'.">
<scol, >
<cbc, >
<out, >
<cstr, "\n">
<scol, >
<out, >
<cstr, "a + b: ">
<scol, >
<out, >
<id, 0>
<sum, >
<id, 1>
<scol, >
<out, >
<cstr, "a - b: ">
<scol, >
<out, >
<id, 0>
<sub, >
<id, 1>
<scol, >
<out, >
<cstr, "\n">
<scol, >
<eof, >
```

Volcado del fichero de la tabla de símbolos

```
Tabla Global #0:
*'a'
*'b'
```

4. **Caso 4:** Funcionamiento erróneo

Código fuente

```
/* Un comentario de bloque sin cerrar es un error léxico, ya que se recibe un EOF inesperado.
```

Errores detectados

```
(2:1) ERROR: Fin de fichero inesperado. Se esperaba «*/» para cerrar el comentario de bloque.
```

5. **Caso 5:** Funcionamiento erróneo

Código fuente

```
/* Hay algunos símbolos por los que un token no puede empezar. */
$ % @ # ?

/* En especial, las variables no pueden empezar con «_». */
var int _error;
```

Errores detectados

```
(2:1) ERROR: Carácter inesperado al buscar el siguiente símbolo («$», U+0024).
(2:3) ERROR: Carácter inesperado al buscar el siguiente símbolo («%», U+0025).
(2:5) ERROR: Carácter inesperado al buscar el siguiente símbolo («@», U+0040).
(2:7) ERROR: Carácter inesperado al buscar el siguiente símbolo («#», U+0023).
(2:9) ERROR: Carácter inesperado al buscar el siguiente símbolo («?», U+003F).
(5:9) ERROR: Carácter inesperado al buscar el siguiente símbolo («_», U+005F).
```

6. **Caso 6:** Funcionamiento erróneo

Código fuente

```
/* Aunque el código esté malformado y no compile, se siguen buscando y generando tokens */
var string a = $ 'Esta cadena se lee correctamente';

/* Esto es útil para depurar los errores sin detenerse sólo en el primero. */
var int a = 2?;

/* En algunos casos, es imposible recuperar tokens en un estado válido y seguir procesando. */
var string s = 'Si no se temina el string, lee todo \'; $$ /**/;; Esto es parte de la cadena.
/* Aquí ya ha terminado la cadena, ya que lee un salto de línea no permitido. */

/* Como la última cadena ya finalizó por error, aquí se recupera y sigue procesando tokens. */
var string s2 = 'Esta cadena se procesa bien';
```

Errores detectados

```
(2:16) ERROR: Carácter inesperado al buscar el siguiente símbolo («$», U+0024).
(5:14) ERROR: Carácter inesperado al buscar el siguiente símbolo («?», U+003F).
(8:94) ERROR: Error en la cadena. Carácter no permitido (U+000D).
```