Práctica 2 - Procesadores de Lenguajes

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

Índice

Introducción

El proyecto a continuación consiste en la construcción de un procesador diseñado para analizar y verificar la corrección léxica, sintáctica y semántica del lenguaje JS–.

El trabajo comenzó con el análisis del lenguaje fuente para reconocer sus elementos fundamentales, lo que nos permitió identificar sus tokens. Gracias a ello, pudimos crear la gramática del lenguaje y su Autómata Finito Determinista equivalente. Con todo esto logramos implementar el Analizador Léxico, así como crear un diseño inicial de la Tabla de Símbolos, un núcleo fundamental para la organización de la información durante el análisis.

A continuación, desarrollamos un Analizador Sintáctico, gracias a la identificación de una nueva gramática que nos permitió corregir la estructura del lenguaje. Finalmente, a esta gramática se integraron las Acciones Semánticas, permitiéndonos realizar la Traducción Dirigida por la Sintaxis propia del Analizador Semántico.

Todo esto fue complementado por un Gestor de Errores claro y conciso, que mejora la experiencia de usuario a la hora de encontrar fallos localizados. El resultado final es un procesador capaz de interpretar un programa escrito en JS–. Esto queda demostrado en los distintos casos de prueba que incluimos.

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 (' ')

Además, hemos decidido usar **C++ como lenguaje de programación** ya que la mayor parte de la infraestructura de compiladores está escrita en C o en C++, incluyendo MSVC (desarrollado por Microsoft) y el proyecto LLVM (utilizado por Google en Android). También hemos tenido en cuenta su flexibilidad y potencia, junto a la amplia variedad de utilidades que tiene su librería estándar. En comparación con lenguajes como Java o JavaScript, C++ ofrece mayor eficiencia y control sobre los recursos del sistema, a la vez que se mantiene versátil y permite escribir código legible.

1. Analizador Léxico

El **Analizador Léxico** constituye la primera etapa en el procesamiento de un lenguaje, pues es el encargado de interactuar directamente con el fichero fuente. Su propósito principal es identificar y clasificar las unidades mínimas del lenguaje, conocidas como **tokens**. Por ello, el primer paso es su identificación.

1.1. Tokens

Para hacer la lista de tokens 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 (ç*")
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).

1.2. 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

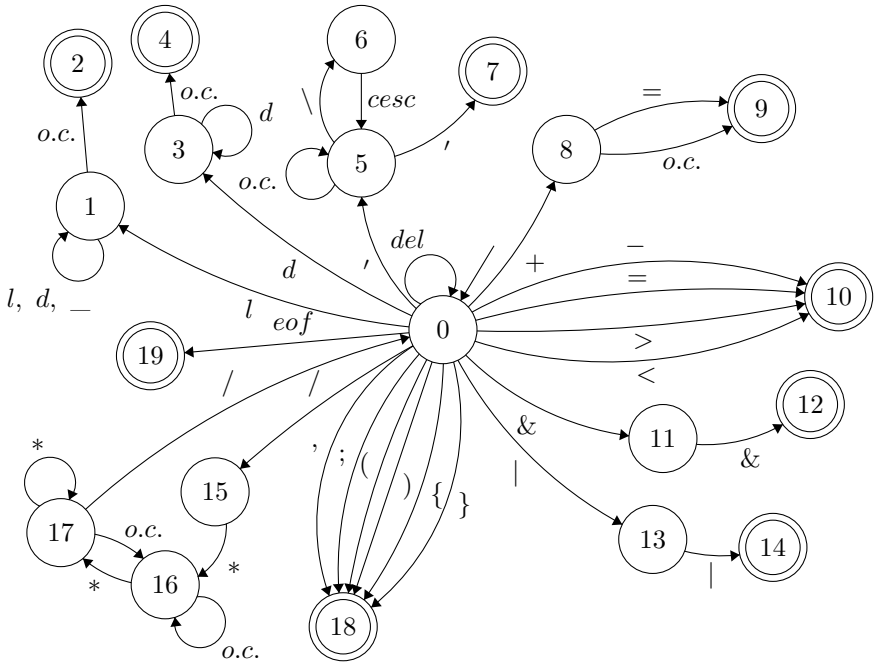
d := 0...9
l := a...z, A...Z
c_1 := espacio o cualquier carácter imprimible menos \
$cesc$:= carácter escapable (', 0, n, a, t, v, f, r, \)
c_2 := cualquier carácter menos * y eof
c_3 := cualquier carácter menos *, / y eof

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

1.3. 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.



1.4. 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

1.5. 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.

2. Analizador Sintáctico

El **Analizador Sintáctico** es la segunda etapa del proceso de análisis y tiene como objetivo principal verificar si la secuencia de tokens generada por el Analizador Léxico cumple con las reglas de la gramática del lenguaje. Su función es comprobar que la estructura del código fuente es válida, de acuerdo con la sintaxis definida para el lenguaje en cuestión. Por ello, recibe los tokens producidos por el Analizador Léxico y construye un árbol sintáctico, que representa la jerarquía y la organización de los elementos del programa. Para llevar a cabo esta tarea utiliza las reglas de una Gramática de Contexto Libre.

2.1. 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* (EXP1) ATOMSTATEMENT
17. STATEMENT → *for* (FORACT ; EXP1 ; FORACT) { BODY }
18. STATEMENT → *var* VARTYPE *id* ;
19. STATEMENT → ATOMSTATEMENT
20. ATOMSTATEMENT → *id* IDACT ;
21. ATOMSTATEMENT → *output* EXP1 ;
22. ATOMSTATEMENT → *input id* ;
23. ATOMSTATEMENT → *return* RETURNEXP ;
24. IDACT → ASS EXP1
25. IDACT → (CALLPARAM)
26. FORACT → *id* ASS EXP1

27. FORACT → λ
28. ASS → =
29. ASS → +=
30. CALLPARAM → EXP1 NEXTPARAM
31. CALLPARAM → λ
32. NEXTPARAM → , EXP1 NEXTPARAM
33. NEXTPARAM → λ
34. RETURNEXP → EXP1
35. RETURNEXP → λ
36. EXP1 → EXP2 EXPOR
37. EXPOR → || EXP2 EXPOR
38. EXPOR → λ
39. EXP2 → EXP3 EXPAND
40. EXPAND → && EXP3 EXPAND
41. EXPAND → λ
42. EXP3 → EXP4 COMP
43. COMP → COMPOP EXP4 COMP
44. COMP → λ
45. COMPOP → >
46. COMPOP → <
47. EXP4 → EXPATOM ARITH
48. ARITH → ARITHOP EXPATOM ARITH
49. ARITH → λ
50. ARITHOP → +
51. ARITHOP → -
52. EXPATOM → *id* IDVAL
53. EXPATOM → (EXP1)
54. EXPATOM → cint
55. EXPATOM → cstr
56. EXPATOM → true
57. EXPATOM → false
58. IDVAL → (CALLPARAM)
59. IDVAL → λ

2.2. 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 → Aα (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, para cada par de reglas A → α | β
- Se cumple que FIRST(α) $\cap$ FIRST(β) = $\emptyset$.
- Si β puede derivar λ, entonces FIRST(α) $\cap$ 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) $\cap$ FIRST(STATEMENT P) = $\emptyset$
FIRST(FUNCTION P) $\cap$ FIRST(*eof*) = $\emptyset$
FIRST(STATEMENT P) $\cap$ FIRST(*eof*) = $\emptyset$

FUNTYPE → *void* | VARTYPE
FIRST(*void*) = {void}
FIRST(VARTYPE) = {int, boolean, string}
FIRST(*void*) $\cap$ FIRST(VARTYPE) = $\emptyset$

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) = ∅

BODY → STATEMENT BODY | λ
FIRST(STATEMENT BODY) = {for, id, if, input, output, return, var}
FIRST(λ) = {λ}
FOLLOW(BODY) = {}}
FIRST(STATEMENT BODY) ∩ FIRST(λ) = ∅
FIRST(STATEMENT BODY) ∩ FOLLOW(BODY) = ∅

Procederíamos de igual manera con el resto de reglas de la gramática con dos o más producciones asociadas, pero dado que el proceso es tedioso y repetitivo, no detallaremos con tanta profundidad en las siguientes comprobaciones:

STATEMENT → *if* (EXP1) ATOMSTATEMENT | *for* (FORACT ; EXP1 ; FORACT) { BODY } |
var VARTYPE *id* ; | ATOMSTATEMENT
FIRST(*if* (EXP1) ATOMSTATEMENT) ∩ FIRST(*for* (FORACT ; EXP1 ; FORACT) { BODY }) = ∅
FIRST(*if* (EXP1) ATOMSTATEMENT) ∩ FIRST(*var* VARTYPE *id* ;) = ∅
FIRST(*if* (EXP1) ATOMSTATEMENT) ∩ FIRST(ATOMSTATEMENT) = ∅
FIRST(*for* (FORACT ; EXP1 ; FORACT) { BODY }) ∩ FIRST(*var* VARTYPE *id* ;) = ∅
FIRST(*for* (FORACT ; EXP1 ; FORACT) { BODY }) ∩ FIRST(ATOMSTATEMENT) = ∅
FIRST(*var* VARTYPE *id* ;) ∩ FIRST(ATOMSTATEMENT) = ∅

ATOMSTATEMENT → *id* IDACT ; | *output* EXP1 ; | *input id* ; | *return* RETURNEXP ;
FIRST(*id* IDACT ;) ∩ FIRST(*output* EXP1 ;) = ∅
FIRST(*id* IDACT ;) ∩ FIRST(*input id* ;) = ∅
FIRST(*id* IDACT ;) ∩ FIRST(*return* RETURNEXP ;) = ∅
FIRST(*output* EXP1 ;) ∩ FIRST(*input id* ;) = ∅
FIRST(*output* EXP1 ;) ∩ FIRST(*return* RETURNEXP ;) = ∅
FIRST(*input id* ;) ∩ FIRST(*return* RETURNEXP ;) = ∅

IDACT → ASS EXP1 | (CALLPARAM)
FIRST(ASS EXP1) ∩ FIRST((CALLPARAM)) = ∅

FORACT → *id* ASS EXP1 | λ
FIRST(*id* ASS EXP1) ∩ FIRST(λ) = ∅
FIRST(*id* ASS EXP1) ∩ FOLLOW(FORACT) = ∅

ASS → = | +=

FIRST(=) ∩ FIRST(+=) = ∅

CALLPARAM → EXP1 NEXTPARAM | λ
FIRST(EXP1 NEXTPARAM) ∩ FIRST(λ) = ∅
FIRST(EXP1 NEXTPARAM) ∩ FOLLOW(CALLPARAM) = ∅

NEXTPARAM → , EXP1 NEXTPARAM | λ
FIRST(, EXP1 NEXTPARAM) ∩ FIRST(λ) = ∅
FIRST(, EXP1 NEXTPARAM) ∩ FOLLOW(NEXTPARAM) = ∅

RETURNEXP → EXP1 | λ
FIRST(EXP1) ∩ FIRST(λ) = ∅
FIRST(EXP1) ∩ FOLLOW(RETURNEXP) = ∅

Repetiríamos el mismo proceso para las reglas restantes:

EXPOR → || EXP2 EXPOR | λ
EXPAND → && EXP3 EXPAND | λ
COMP → COMPOP EXP4 COMP | λ
COMPOP → >| <
ARITH → ARITHOP EXPATOM ARITH | λ
ARITHOP → + | -
EXPATOM → *id* IDVAL | (EXP) | *cint* | *cstr* | *true* | *false*
IDVAL → (CALLPARAM) | λ

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 sección de herramientas de la página web del departamento.

2.3. Pseudo-código con funciones del Analizador Sintáctico

En esta sección, presentamos el diseño en pseudo-código de las funciones que conforman el Analizador Sintáctico. Para cada símbolo no terminal de la gramática, se implementa una función que sigue un esquema de if-then-else anidado, donde cada rama corresponde a una posible regla. El token recibido desde el Analizador Léxico determina la rama que se ejecuta, iniciando el recorrido del consecuente de la regla seleccionada. Para cada símbolo del consecuente:

- Si resulta ser un terminal, se equipara con el token actual. En el caso de que coincidan, se le solicita un nuevo token al Analizador Léxico y, si no, se genera un error sintáctico.
- Si es un no terminal, se realiza una llamada recursiva a la función correspondiente. El main del Analizador Sintáctico inicia con la solicitud del primer token y llamando a la función del axioma.

Si al terminar esta función la cadena se ha procesado por completo, el análisis concluye con éxito. En caso contrario, se reporta un error sintáctico. En el anexo B presentamos la estructura de dichas funciones.

3. Analizador Semántico

El **Analizador Semántico** es la tercera etapa del procesamiento de un lenguaje, cuyo objetivo es asegurar que el programa cumpla con las reglas y las expectativas semánticas del lenguaje. Una vez que el código ha sido analizado léxica y sintácticamente, el Analizador Semántico se encarga de verificar que las operaciones y relaciones entre los elementos del programa sean lógicas y coherentes.

3.1. Traducción Dirigida por la Sintaxis

Para realizar este análisis semántico se debe llevar a cabo la **Traducción Dirigida por la Sintaxis**. La hemos representado mediante un **Esquema de Traducción**, para lo que se deben añadir las **Acciones Semánticas** directamente a la gramática del Analizador Sintáctico.

0. P' → **{TSG := CrearTS(); TSL := NULL; DespG := 0}** P **{DestruirTS(TSG)}**
1. P → FUNCTION P
2. P → STATEMENT P
3. P → *eof*
4. FUNCTION → function FUNTYPE *id* **{TSL := CrearTS(); DespL := 0}**
   (FUNATTRIBUTES) **{InsertaTipoTS(id.pos,**
   FUNATTRIBUTES.tipo → FUNTYPE.tipo);
   InsertaEtiquetaTS(id.pos, nueva_etiqueta())}
   { BODY } **{if (BODY.tipo != tipo_ok) then error(000)**
   if (FUNTYPE.tipo != BODY.tipoRet)
   then error(101)
   DestruirTS(TSL)}
5. FUNTYPE → *void* **{FUNTYPE.tipo := void}**
6. FUNTYPE → VARTYPE **{FUNTYPE.tipo := VARTYPE.tipo}**
7. VARTYPE → *int* **{VARTYPE.tipo := int; := VARTYPE.ancho := 1}**
8. VARTYPE → *boolean* **{VARTYPE.tipo := log; := VARTYPE.ancho := 1}**
9. VARTYPE → *string* **{VARTYPE.tipo := str; := VARTYPE.ancho := 64}**
10. FUNATTRIBUTES → *void* **{FUNATTRIBUTES.tipo := void}**
11. FUNATTRIBUTES → VARTYPE *id* **{ if(BuscaTipoTS(id.pos) != NULL)**
    then error(000)
    {InsertaTipoTS(id.pos, VARTYPE.tipo);
    InsertaDespTS(id.pos, DespL);
    DespL := DespL + VARTYPE.tipo}
    NEXTATTRIBUTE **{FUNATTRIBUTES.tipo :=**
    if (NEXTATTRIBUTE.tipo != void) then
    VARTYPE.tipo × NEXTATTRIBUTE.tipo
    else VARTYPE.tipo}
12. NEXTATTRIBUTE → , VARTYPE *id* **{ if(BuscaTipoTS(id.pos) != NULL)**
    then error(000)
    {InsertaTipoTS(id.pos, VARTYPE.tipo);
    InsertaDespTS(id.pos, DespL);
    DespL := DespL + VARTYPE.tipo}
    NEXTATTRIBUTE_1 **{NEXTATTRIBUTE.tipo :=**
    if (NEXTATTRIBUTE_1.tipo != void) then
    VARTYPE.tipo × NEXTATTRIBUTE_1.tipo
    else VARTYPE.tipo}
13. NEXTATTRIBUTE → λ **{NEXTATTRIBUTE.tipo := void}**
14. BODY → STATEMENT BODY_1
    {BODY.tipo := if (STATEMENT.tipo = tipo_ok)
    then BODY_1.tipo
    else tipo_error
    BODY.tipoRet := if(STATEMENT.tipoRet = BODY_1.tipoRet
    or STATEMENT.tipoRet = void) then
    BODY_1.tipoRet
    else if (BODY_1.tipoRet = void) then
    STATEMENT.tipoRet
    else tipo_error}
15. BODY → λ **{BODY.tipo := tipo_ok; BODY.tipoRet := void}**
16. STATEMENT → *if* (EXP1) ATOMSTATEMENT

{STATEMENT.tipo := if (EXP1.tipo != log)
then tipo_error
else ATOMSTATEMENT.tipo
STATEMENT.tipoRet := ATOMSTATEMENT.tipoRet}

17. STATEMENT → *for* (FORACT$_1$; EXP1 ; FORACT$_2$) { BODY }
{STATEMENT.tipo := if (FORACT$_1$.tipo = tipo_ok
and EXP1.tipo = log and FORACT$_2$.tipo = tipo_ok)
then BODY.tipo
else error(103)
STATEMENT.tipoRet := BODY.tipoRet}
18. STATEMENT → *var* VARTYPE *id* ; **{if(BuscaTipoTS(id.pos) != NULL)**
then error(000)
insertaTipoTS(id.pos, VARTYPE.tipo)
if(TSL != NULL) then
insertaDespTS(id.pos,despG)
despG := despG + VARTYPE.ancho
else
insertaDespTS(id.pos,despL)
despL := despL + VARTYPE.ancho
STATEMENT.tipo := tipo_ok}
19. STATEMENT → ATOMSTATEMENT **{STATEMENT.tipo := ATOMSTATEMENT.tipo**
STATEMENT.tipoRet := ATOMSTATEMENT.tipoRet}
20. ATOMSTATEMENT → *id* IDACT ; **{if(IDACT.funCall){**
if (id.tipo != R → T) then
ATOMSTATEMENT.tipo := tipo_error
error(201)
else if (IDACT.tipo = tipo_error)
then ATOMSTATEMENT = tipo_error
else if (IDACT.tipo = R.tipo)
then ATOMSTATEMENT = tipo_ok
else
ATOMSTATEMENT = tipo_error
error(203)
} else {
if (id.tipo = R → T) then
ATOMSTATEMENT.tipo := tipo_error
error(202)
else if (IDACT.tipo = tipo_error)
then ATOMSTATEMENT = tipo_error
else if (IDACT.tipo = R.tipo)
then ATOMSTATEMENT = tipo_ok
else
ATOMSTATEMENT = tipo_error
error(200)
21. ATOMSTATEMENT → *output* EXP1 ; **{if (EXP1.tipo ∈ {int, str}) then**
ATOMSTATEMENT.tipo := tipo_ok
else
ATOMSTATEMENT.tipo := tipo_error
error(104)
ATOMSTATEMENT.tipoRet := void}
22. ATOMSTATEMENT → *input id* ; **{if (buscaTipoTS(id.pos) ∈ {int, str})**
then ATOMSTATEMENT.tipo := tipo_ok

else
ATOMSTATEMENT.tipo := tipo_error
error(105)
ATOMSTATEMENT.tipoRet := void}

23. ATOMSTATEMENT → *return* RETURNEXP ; **{ATOMSTATEMENT.tipo :=**
if (RETURNEXP.tipo != tipo_error) then tipo_ok
else tipo_error
ATOMSTATEMENT.tipoRet := RETURNEXP.tipo}

24. IDACT → ASS EXP1 **{IDACT.tipo :=**
if ((ASS.sum = true AND EXP1.tipo ∈ {int, str})
OR ASS.sum = false) then EXP1.tipo
else tipo_error}
25. IDACT → (CALLPARAM) **{IDACT.tipo := CALLPARAM.tipo}**
26. FORACT → *id* ASS EXP1 **{if (buscaTipoTS(id.pos) != int)**
FORACT.tipo := tipo_error
error(102)
else if EXP1.tipo = tipo_error
FORACT.tipo := tipo_error
else if EXP1.tipo != int
FORACT.tipo := tipo_error
error(200)
else FORACT.tipo := tipo_ok
27. FORACT → λ **{FORACT.tipo := tipo_ok}**
28. ASS → = **{ASS.sum = false}**
29. ASS → += **{ASS.sum = true}**
30. CALLPARAM → EXP1 NEXTPARAM **{CALLPARAM.tipo :=**
if (EXP1.tipo = tipo_error
OR NEXTPARAM.tipo == tipo_error) then
tipo_error
else if (NEXTPARAM.tipo != void) then
EXP1.tipo × NEXTPARAM.tipo
else tipo_error}
31. CALLPARAM → λ **{CALLPARAM.tipo := void}**
32. NEXTPARAM → , EXP1 NEXTPARAM_1 **{CALLPARAM.tipo :=**
if (EXP1.tipo = tipo_error
OR NEXTPARAM_1.tipo == tipo_error) then
tipo_error
else if (NEXTPARAM_1.tipo != void) then
EXP1.tipo × NEXTPARAM_1.tipo
else tipo_error}
33. NEXTPARAM → λ **{NEXTPARAM.tipo := void}**
34. RETURNEXP → EXP1 **RETURNEXP.tipo := EXP1.tipo**
35. RETURNEXP → λ **RETURNEXP.tipo := void**
36. EXP1 → EXP2 EXPOR **{if (EXPOR.tipo = void) then EXP1.tipo := EXP2.tipo**
else if (EXP2.tipo != log OR EXPOR.tipo = tipo_error) then
error(100)
EXP1.tipo := tipo_error
else EXP1.tipo := log}
37. EXPOR → || EXP2 EXPOR_1
{if (EXP2.tipo != log OR EXPOR_1.tipo = tipo_error) then
error(100)
EXPOR.tipo := tipo_error
else EXPOR.tipo := logico}

38. EXPOR → λ **{EXPOR.tipo := void}**
39. EXP2 → EXP3 EXPAND
{if (EXPAND.tipo = void) then EXP2.tipo := EXP3.tipo
else if (EXP3.tipo != log OR EXPAND.tipo = tipo_error) then
error(100)
EXP2.tipo := tipo_error
else EXP2.tipo := log}
40. EXPAND → && EXP3 $EXPAND_1$
{if (EXP3.tipo != log OR $EXPAND_1$.tipo = tipo_error) then
error(100)
EXPAND.tipo := tipo_error
else EXPAND.tipo := logico}
41. EXPAND → λ **{EXPAND.tipo := void}**

42. EXP3 → EXP4 COMP **{if (COMP.tipo = void) then EXP3.tipo := EXP4.tipo**
else if (EXP4.tipo != int OR COMP.tipo = tipo_error) then
error(100)
EXP3.tipo := tipo_error
else EXP3.tipo := log}
43. COMP → COMPOP EXP4 $COMP_1$
{if (EXP4.tipo != int OR $COMP_1$.tipo = tipo_error) then
error(100)
COMP.tipo := tipo_error
else COMP.tipo := logico}
44. COMP → λ **{COMP.tipo := void}**
45. COMPOP → >
46. COMPOP → <
47. EXP4 → EXPATOM ARITH
{if (ARITH.tipo = void) then EXP4.tipo := EXPATOM.tipo
{else if (EXPATOM.tipo != int AND EXPATOM.tipo != string) then
EXP4.tipo := tipo_error
error(100)
else if (ARITH.tipo = tipo_error) then EXP4.tipo := tipo_error
else if (EXPATOM != ARITH.tipo) then
EXP4.tipo := tipo_error
error(200)
else EXP4.tipo := ARITH.tipo}
48. ARITH → ARITHOP EXPATOM $ARITH_1$
{if (ARITHOP.sum = false AND EXPATOM.tipo != int) then
ARITH.tipo := tipo_error
error(100)
else if (ARITHOP.sum = true AND EXPATOM.tipo != int
AND EXPATOM.tipo != string) then
ARITH.tipo := tipo_error
error(100)
else if ($ARITH_1$.tipo = tipo_error)
ARITH.tipo := tipo_error
else if ($ARITH_1$.tipo != void AND EXPATOM.tipo != $ARITH_1$.tipo) then
ARITH.tipo := tipo_error
error(200)
else ARITH.tipo := EXPATOM.tipo}
49. ARITH → λ **{ARITH.tipo := void}**
50. ARITHOP → +
51. ARITHOP → -

52. EXPATOM → *id* IDVAL **{if(IDVAL.funCall){**
if (id.tipo != R → T) then
EXPATOM.tipo := tipo_error
error(201)
else if (IDVAL.tipo = tipo_error)
then EXPATOM = tipo_error
else if (IDVAL.tipo = R.tipo)
then EXPATOM = T.tipo
else
EXPATOM = tipo_error
error(203)
} else {
if (id.tipo = R → T) then
EXPATOM.tipo := tipo_error
error(202)
else
EXPATOM = id.tipo
53. EXPATOM → (EXP1) **{EXPATOM.tipo := EXP1.tipo}**
54. EXPATOM → cint **{EXPATOM.tipo := int}**
55. EXPATOM → cstr **{EXPATOM.tipo := str}**
56. EXPATOM → true **{EXPATOM.tipo := log}**
57. EXPATOM → false **{EXPATOM.tipo := log}**
58. IDVAL → (CALLPARAM)
59. IDVAL → λ

4. Tabla de Símbolos

La Tabla de Símbolos (TS) es una estructura de datos fundamental en la implementación de un compilador o procesador de lenguajes. Su principal objetivo es **almacenar información sobre los identificadores** (variables, funciones, etc.) que aparecen en el programa fuente y organizarla de manera eficiente para su consulta durante las fases de análisis semántico y de ejecución. En nuestro caso, hemos implementado una TS lineal y dinámica, que gestiona tanto la información de los identificadores como los alcances de los mismos.

4.1. Descripción de su estructura

La Tabla de Símbolos Global es esencialmente una colección de entradas, donde cada entrada corresponde a un símbolo (un identificador). Cada símbolo tiene varios atributos dependiendo de su tipo (por ejemplo, una variable, una función, un array, etc.). Sin embargo, este proyecto requiere de la existencia de Tablas Locales. Estas almacenan información sobre los identificadores dentro de un alcance específico.

Es por esta razón que nuestra Tabla de Símbolos tiene una estructura de pila de tablas de símbolos. Cuando se ingresa a un nuevo bloque de código (por ejemplo, al ingresar a una función), se puede crear una nueva tabla local que se apila encima de la tabla global o de las tablas locales anteriores. Cuando se sale del bloque, se elimina la tabla local, y el procesador de lenguajes vuelve a la tabla local anterior o a la global.

Nuestra estructura principal de la Tabla de Símbolos incluye:

- **Struct Symbol**: Cada símbolo contiene la siguiente información:
  - **Lexema**: nombre del identificador
  - **Atributos específicos**: que pueden ser cualquier par clave-valor, como tipo, desplazamiento, número de parámetros, etc.
  - **Tipo de identificador** (opcional): puede agregarse para especificar el tipo del identificador

- **Clase SymbolTables**: La clase principal que gestiona las tablas de símbolos:
  - **Clase Table**: Una tabla de símbolos que contiene:
    - **Identificador de tabla**: único para cada tabla
    - **Lista de símbolos**: vector con los símbolos de la tabla.
    - **Mapa de nombres a posiciones**: un mapa que asocia los nombres de los símbolos con sus posiciones en el vector.
    - **Métodos**:
      - **AddSymbol**: Agrega un nuevo símbolo a la tabla y retorna su posición.
      - **AddAttribute**: Agrega un atributo al símbolo en la posición indicada.
      - **SearchSymbol**: Busca un símbolo por su nombre en la tabla y devuelve su posición si lo encuentra.
      - **WriteTable**: Escribe el contenido de la tabla por consola.
  - **Contador de tablas**: para asignar un identificador único a cada tabla creada.
  - **Lista de tablas**: un vector que contiene las tablas de símbolos en orden jerárquico.
  - **Métodos para gestionar tablas y símbolos**:
    - **CreateTable()**: Crea una nueva tabla de símbolos y la agrega a la lista de tablas.
    - **DestroyTable()**: Elimina la tabla de símbolos más reciente de la lista de tablas.
    - **AddSymbol()**: Agrega un nuevo símbolo a la tabla más reciente, dado su nombre.
    - **AddGlobalSymbol()**: Agrega un símbolo a la tabla global.
    - **AddAttribute()**: Agrega un atributo a un símbolo específico en la tabla más reciente, dado su índice, el nombre del atributo y su valor.
    - **SearchSymbol()**: Busca un símbolo en todas las tablas (empezando desde la tabla más reciente) y devuelve su posición si es encontrado.
    - **WriteTable()**: Escribe la tabla de símbolos más reciente por consola.

4.2. Organización

En cuanto a la organización de la tabla de símbolos, se trata de un mecanismo que permite estructurar y gestionar los identificadores en un programa de manera eficiente, para que se pueda acceder a ellos correctamente.

1. **Tablas Globales y Locales:**
   - La **tabla global** contiene símbolos con alcance global, es decir, aquellos que pueden ser utilizados en cualquier parte del programa.
   - Las **tablas locales** contienen símbolos con alcance local, que solo son accesibles dentro de la función o bloque donde se han declarado.
2. **Estructura de las Tablas:**
   - Cada tabla de símbolos (global o local) puede organizarse como una lista de entradas por el nombre del símbolo.
   - Cada entrada en la tabla contiene información relevante sobre el símbolo, como su nombre, tipo, atributos adicionales, desplazamiento en memoria, o tipo de retorno en caso de funciones.
3. **Organización Jerárquica:**
   - Las tablas de símbolos se organizan en una **jerarquía**. Las tablas locales pueden referirse a símbolos definidos en tablas globales.
4. **Relación entre Tablas:**

- Las tablas locales y globales se relacionan para gestionar el alcance de los símbolos.
- Los símbolos locales y globales pueden compartir nombres sin interferir, ya que se encuentran en tablas separadas, lo que evita conflictos de nombres.
- Para **acceder a un símbolo**, el programa primero consulta la tabla local asociada al bloque o función actual. Si el símbolo no se encuentra allí, se consulta la tabla global.

5. Diseño del Gestor de Errores

6. Demostración del funcionamiento

A. Anexo - Analizador Léxico

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).
```

B. Anexo - Pseudocódigo

Función 1: Main del Analizador Sintáctico

```
Function A_Sint() {
    sig_tok := ALex();
    P;
    if sig_tok ≠ '$' then error();
}
```

Función 2: Equipara

```
Function equipara (t){
    if sig_tok == t
    then sig_tok := ALex()
    else error ()
}
```

Función 3: P

```
Function P() {
    if sig_tok == 'function' then {
        print(1);
        FUNCTION();
        P();
    }
    else if sig_tok ∈ {'for', 'id', 'if', 'input', 'output', 'return', 'var'} then {
        print(2);
        STATEMENT();
        P();
    }
    else if sig_tok == 'eof' then {
        print(3);
        equipara( eof );
    }
    else error();
}
```

Función 4: FUNCTION

```
Function FUNCTION() {
    if sig_tok == 'function' then {
        print(4);
        equipara(function);
        FUNTYPE();
        equipara(id);
        equipara(();
        FUNATTRIBUTES();
        equipara());
        equipara({);
        BODY();
        equipara(});
    }
    else error();
}
```

Función 5: FUNTYPE

```
Function FUNTYPE() {
    if sig_tok == 'void' then {
        print(5);
        equipara(void);
    }
    else if sig_tok ∈ {'boolean', 'int', 'string'} then {
        print(6);
        VARTYPE();
    }
    else error();
}
```

Función 6: VARTYPE

```
Function VARTYPE() {
    if sig_tok == 'int' then {
        print(7);
        equipara(int);
    }
    else if sig_tok == 'boolean' then {
        print(8);
        equipara(boolean);
    }
    else if sig_tok == 'string' then {
        print(9);
        equipara(string);
    }
    else error();
}
```

Función 7: FUNATTRIBUTES

```
Function FUNATTRIBUTES() {
    if sig_tok == 'void' then {
        print(10);
        equipara(void);
    }
    else if sig_tok ∈ {'boolean', 'int', 'string'} then {
        print(11);
        VARTYPE();
        equipara(id);
        NEXTATTRIBUTE();
    }
    else error();
}
```

Función 8: NEXTATTRIBUTE

```
Function NEXTATTRIBUTE() {
    if sig_tok == ',' then {
        print(12);
        equipara(,);
        VARTYPE();
        equipara(id);
        NEXTATTRIBUTE();
    }
    else if sig_tok = ')' then
        print(13);
    else error();
}
```

Función 9: BODY

```
Function BODY() {
    if sig_tok ∈ {'for', 'id', 'if', 'input', 'output', 'return', 'var'} then {
        print(14);
        STATEMENT();
        BODY();
    }
    else if sig_tok = '}' then
        print(15);
    else error();
}
```

Función 10: ATOMSTATEMENT

```
Function ATOMSTATEMENT() {
    if sig_tok == 'id' then {
        print(16);
        equipara(id);
        IDACT();
        equipara( ; );
    }
    else if sig_tok == 'output' then {
        print(17);
        equipara(output);
        EXP();
        equipara( ; );
    }
    else if sig_tok == 'input' then {
        print(18);
        equipara(input);
        equipara(id);
        equipara( ; );
    }
    else if sig_tok == 'return' then {
        print(19);
        equipara(return);
        RETURNEXP();
        equipara( ; );
    }
    else error();
}
```

Función 11: IDACT

```
Function IDACT() {
    if sig_tok ∈ {'=', '+='} then {
        print(20);
        ASS();
        EXP();
    }
    else if sig_tok == '(' then {
        print(21);
        equipara(();
        CALLPARAM();
        equipara());
    }
    else error();
}
```

Función 12: FORACT

```
Function FORACT() {
    if sig_tok == 'id' then {
        print(22);
        equipara(id);
        ASS();
        EXP();
    }
    else if sig_tok ∈ {')', ';'} then {
        print(23);
    }
    else error();
}
```

Función 13: ASS

```
Function ASS() {
    if sig_tok == '=' then {
        print(24);
        equipara(=);
    }
    else if sig_tok == '+=' then {
        print(25);
        equipara(+=);
    }
    else error();
}
```

Función 14: CALLPARAM

```
Function CALLPARAM() {
    if sig_tok ∈ {'(', 'cint', 'cstr', 'false', 'id', 'true', 'lambda'} then {
        print(26);
        EXP();
        NEXTPARAM();
    }
    else if sig_tok == ')' then {
        print(27);
    }
    else error();
}
```

Función 15: NEXTPARAM

```
Function NEXTPARAM() {
    if sig_tok == ',' then {
        print(28);
        equipara(,);
        EXP();
        NEXTPARAM();
    }
    else if sig_tok == ')' then {
        print(29);
    }
    else error();
}
```

Función 16: RETURNEXP

```
Function RETURNEXP() {
    if sig_tok ∈ {'(', 'cint', 'cstr', 'false', 'id', 'true'} then {
        print(30);
        EXP();
    }
    else if sig_tok == ';' then {
        print(31);
    }
    else error();
}
```

Función 17: EXP

```
Function EXP() {
    print(32);
    A();
    EXP1();
}
```

Función 18: EXP1

```
Function EXP1() {
    if sig_tok ∈ {'&&', '||'} then {
        print(33);
        LOGOP();
        A();
        EXP1();
    }
    else if sig_tok ∈ {')', ';', ','} then {
        print(34);
    }
    else error();
}
```

Función 19: LOGOP

```
Function LOGOP() {
    if sig_tok == '&&' then {
        print(35);
        equipara(&&);
    }
    else if sig_tok == '||' then {
        print(36);
        equipara(||);
    }
    else error();
}
```

Función 20: A

```
Function A() {
    print(37);
    B();
    A1();
}
```

Función 21: A1

```
Function A1() {
    if sig_tok ∈ {'>', '<'} then {
        print(38);
        COMPOP();
        B();
        A1();
    }
    else if sig_tok ∈ {'&&', '||', ')', ';', ','} then {
        print(39);
    }
    else error();
}
```

Función 22: COMPOP

```
Function COMPOP() {
    if sig_tok == '>' then {
        print(40);
        equipara(>);
    }
    else if sig_tok == '<' then {
        print(41);
        equipara(<);
    }
    else error();
}
```

Función 23: B

```
Function B() {
    print(42);
    EXPATOM();
    B1();
}
```

Función 24: B1

```
Function B1() {
    if sig_tok ∈ {'+', '-'} then {
        print(43);
        ARITHMETICOP();
        EXPATOM();
        B1();
    }
    else if sig_tok ∈ {'&&', '||', ')', ';', ',', '<', '>'} then {
        print(44);
    }
    else error();
}
```

Función 25: ARITHMETICOP

```
Function ARITHMETICOP() {
    if sig_tok == '+' then {
        print(45);
        equipara(+);
    }
    else if sig_tok == '-' then {
        print(46);
        equipara(-);
    }
    else error();
}
```

Función 26: EXPATOM

```
Function EXPATOM() {
    if sig_tok == 'id' then {
        print(47);
        equipara(id);
        C();
    }
    else if sig_tok == '(' then {
        print(48);
        equipara(();
        EXP();
        equipara());
    }
    else if sig_tok == 'cint' then {
        print(49);
        equipara(cint);
    }
    else if sig_tok == 'cstr' then {
        print(50);
        equipara(cstr);
    }
    else if sig_tok == 'true' then {
        print(51);
        equipara(true);
    }
    else if sig_tok == 'false' then {
        print(52);
        equipara(false);
    }
    else error();
}
```

Función 27: C

```
Function C() {
    if sig_tok == '(' then {
        print(53);
        equipara(();
        CALLPARAM();
        equipara());
    }
    else if sig_tok ∈ {'&&', '||', ')', ';', ',', '<', '>', '+','-'} then {
        print(54);
    }
    else error();
}
```

C. Anexo - Analizador Sintáctico

1. **Caso 1:** Funcionamiento correcto

Código fuente

```
/* Programa con con sólo declaraciones de nivel superior. */
var boolean a;
var int b;
var string c;

/* Declarar de nuevo una variable es error semántico, no sintáctico. */
var int a;
```

Volcado del fichero de parse

Des 2 18 8 2 18 7 2 18 9 2 18 7 3

Árbol sintáctico generado con la herramienta VASt

```
· P (2)
  · STATEMENT (18)
    · var
    · VARTYPE (8)
      · boolean
    · id
    · ;
  · P (2)
    · STATEMENT (18)
      · var
      · VARTYPE (7)
        · int
      · id
      · ;
    · P (2)
      · STATEMENT (18)
        · var
        · VARTYPE (9)
          · string
        · id
        · ;
      · P (2)
        · STATEMENT (18)
          · var
          · VARTYPE (7)
            · int
          · id
          · ;
        · P (3)
          · eof
```

2. **Caso 2:** Funcionamiento correcto

Código fuente

```
/* En este programa, definimos y llamamos a funciones. */

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 parse

Des 1 4 5 11 9 13 14 19 21 36 41 46 51 58 48 43 38 14 19 21 36 41 46 54 48 43 38 15 2 19 20 25 30 36 41 46 54 48 43 38 33 2 19 20 25 30 36 41 46 54 48 43 38 33 3

Árbol sintáctico generado con la herramienta VASt

```
· P (1)
  · FUNCTION (4)
    · function
    · FUNTYPE (5)
      · void
    · id
    · (
    · FUNATTRIBUTES (11)
      · VARTYPE (9)
        · string
      · id
      · NEXTATTRIBUTE (13)
        · lambda
    · )
    · {
    · BODY (14)
      · STATEMENT (19)
        · ATOMSTATEMENT (21)
          · output
          · EXP (36)
            · A (41)
              · B (46)
                · EXPATOM (51)
                  · id
                  · C (58)
                    · lambda
                · B1 (48)
                  · lambda
              · A1 (43)
                · lambda
            · EXP1 (38)
              · lambda
          · ;
      · BODY (14)
        · STATEMENT (19)
          · ATOMSTATEMENT (21)
            · output
            · EXP (36)
              · A (41)
                · B (46)
                  · EXPATOM (54)
                    · cstr
                  · B1 (48)
                    · lambda
                · A1 (43)
                  · lambda
              · EXP1 (38)
                · lambda
            · ;
        · BODY (15)
          · lambda
    · }
  · P (2)
    · STATEMENT (19)
      · ATOMSTATEMENT (20)
        · id
        · IDACT (25)
```

```
          · (
          · CALLPARAM (30)
            · EXP (36)
              · A (41)
                · B (46)
                  · EXPATOM (54)
                    · cstr
                  · B1 (48)
                    · lambda
                · A1 (43)
                  · lambda
              · EXP1 (38)
                · lambda
            · NEXTPARAM (33)
              · lambda
          · )
        · ;
    · P (2)
      · STATEMENT (19)
        · ATOMSTATEMENT (20)
          · id
          · IDACT (25)
            · (
            · CALLPARAM (30)
              · EXP (36)
                · A (41)
                  · B (46)
                    · EXPATOM (54)
                      · cstr
                    · B1 (48)
                      · lambda
                  · A1 (43)
                    · lambda
                · EXP1 (38)
                  · lambda
              · NEXTPARAM (33)
                · lambda
            · )
          · ;
      · P (3)
        · eof
```

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 parse

Des 2 19 22 2 19 22 2 16 36 41 46 51 58 48 42 45 46 51 58 48 43 38 21 36 41 46 54 48 43 38 2 16 36 41 46 51 58 48 42 44 46 51 58 48 43 38 21 36 41 46 54 48 43 38 2 19 21 36 41 46 54 48 43 38 2 19 21 36 41 46 54 48 43 38 2 19 21 36 41 46 51 58 47 49 51 58 48 43 38 2 19 21 36 41 46 54 48 43 38 2 19 21 36 41 46 51 58 47 50 51 58 48 43 38 2 19 21 36 41 46 54 48 43 38 3

Árbol sintáctico generado con la herramienta VASt

```
· P (2)
  · STATEMENT (19)
    · ATOMSTATEMENT (22)
      · input
      · id
      · ;
  · P (2)
    · STATEMENT (19)
      · ATOMSTATEMENT (22)
        · input
        · id
        · ;
    · P (2)
      · STATEMENT (16)
        · if
        · (
        · EXP (36)
          · A (41)
            · B (46)
              · EXPATOM (51)
                · id
                · C (58)
                  · lambda
              · B1 (48)
                · lambda
            · A1 (42)
              · COMPOP (45)
                · 
              · B (46)
                · EXPATOM (51)
                  · id
                  · C (58)
                    · lambda
                · B1 (48)
                  · lambda
              · A1 (43)
                · lambda
          · EXP1 (38)
            · lambda
        · )
        · ATOMSTATEMENT (21)
          · output
          · EXP (36)
            · A (41)
              · B (46)
                · EXPATOM (54)
                  · cstr
                · B1 (48)
                  · lambda
              · A1 (43)
                · lambda
            · EXP1 (38)
              · lambda
          · ;
      · P (2)
        · STATEMENT (16)
          · if
          · (
          · EXP (36)
```

```
            · A (41)
              · B (46)
                · EXPATOM (51)
                  · id
                  · C (58)
                    · lambda
                · B1 (48)
                  · lambda
              · A1 (42)
                · COMPOP (44)
                  · 
                · B (46)
                  · EXPATOM (51)
                    · id
                    · C (58)
                      · lambda
                  · B1 (48)
                    · lambda
                · A1 (43)
                  · lambda
            · EXP1 (38)
              · lambda
          · )
          · ATOMSTATEMENT (21)
            · output
            · EXP (36)
              · A (41)
                · B (46)
                  · EXPATOM (54)
                    · cstr
                  · B1 (48)
                    · lambda
                · A1 (43)
                  · lambda
              · EXP1 (38)
                · lambda
            · ;
        · P (2)
          · STATEMENT (19)
            · ATOMSTATEMENT (21)
              · output
              · EXP (36)
                · A (41)
                  · B (46)
                    · EXPATOM (54)
                      · cstr
                    · B1 (48)
                      · lambda
                  · A1 (43)
                    · lambda
                · EXP1 (38)
                  · lambda
              · ;
            · P (2)
              · STATEMENT (19)
                · ATOMSTATEMENT (21)
                  · output
                  · EXP (36)
                    · A (41)
                      · B (46)
                        · EXPATOM (54)
                          · cstr
                        · B1 (48)
                          · lambda
                      · A1 (43)
                        · lambda
                    · EXP1 (38)
                      · lambda
                  · ;
                · P (2)
                  · STATEMENT (19)
```

```
                · ATOMSTATEMENT (21)
                  · output
                  · EXP (36)
                    · A (41)
                      · B (46)
                        · EXPATOM (51)
                          · id
                          · C (58)
                            · lambda
                        · B1 (47)
                          · ARITHMETICOP (49)
                            · +
                          · EXPATOM (51)
                            · id
                            · C (58)
                              · lambda
                          · B1 (48)
                            · lambda
                      · A1 (43)
                        · lambda
                    · EXP1 (38)
                      · lambda
                  · ;
              · P (2)
                · STATEMENT (19)
                  · ATOMSTATEMENT (21)
                    · output
                    · EXP (36)
                      · A (41)
                        · B (46)
                          · EXPATOM (54)
                            · cstr
                          · B1 (48)
                            · lambda
                        · A1 (43)
                          · lambda
                      · EXP1 (38)
                        · lambda
                    · ;
                · P (2)
                  · STATEMENT (19)
                    · ATOMSTATEMENT (21)
                      · output
                      · EXP (36)
                        · A (41)
                          · B (46)
                            · EXPATOM (51)
                              · id
                              · C (58)
                                · lambda
                            · B1 (47)
                              · ARITHMETICOP (50)
                                · -
                              · EXPATOM (51)
                                · id
                                · C (58)
                                  · lambda
                              · B1 (48)
                                · lambda
                          · A1 (43)
                            · lambda
                        · EXP1 (38)
                          · lambda
                      · ;
                  · P (2)
                    · STATEMENT (19)
                      · ATOMSTATEMENT (21)
                        · output
                        · EXP (36)
                          · A (41)
                            · B (46)
```

```
                        · EXPATOM (54)
                          · cstr
                        · B1 (48)
                          · lambda
                      · A1 (43)
                        · lambda
                    · EXP1 (38)
                      · lambda
                  · ;
              · P (3)
                · eof
```

4. **Caso 4:** Funcionamiento erróneo

Código fuente

```
/* No podemos declarar una variable con un tipo inexistente. */
var no_type err;
```

Errores detectados

```
(2:12) ERROR: Tipo de variable desconocido.
```

5. **Caso 5:** Funcionamiento erróneo

Código fuente

```
/* Una función puede usa «void» si no devuelve nada o si no toma argumentos. */
function void empty(void) {}

/* Una variable no puede ser de tipo «void». */
var void err;
```

Errores detectados

```
(5:9) ERROR: Una variable no puede ser de tipo «void».
```

6. **Caso 6:** Funcionamiento erróneo

Código fuente

```
/* Usar un identificador dos veces es error semántico, no sintáctico. */
var int a;

function int a(string arg) {
    output arg
    /* El sintáctico no verifica que se devuelva un valor valido. */
};

/* Podemos realizar varias acciones sobre identificadores. */

/* Asignar un valor, si es una variable. */
a = a + 3;

/* Realizar una llamada, si es función. */
result = a();
```

```
/* No realizar ninguna acción es inválido. */
a;
```

Errores detectados

```
(7:2) ERROR: Expresión incorrecta: Se esperaba alguna acción sobre el identificador.
```