

Procesadores de Lenguaje – 2019/2020

Creación de un compilador

Práctica 3

Jesús Andrés Fernández
Luna Jiménez Fernández

Índice

1. Introducción.....	4
2. Analizador léxico	4
2.1. Especificación formal del lenguaje	4
2.2. Máquina Discriminadora Determinista (MDD) asociada.....	5
2.3. Implementación	6
2.4. Ejemplos	7
3. Analizador sintáctico	9
3.1. Producciones del lenguaje	9
3.2. Comprobación y transformación de la gramática a LL(1)	10
3.3. Primeros y siguientes de la gramática	10
3.3.1. Primeros.....	10
3.3.2. Siguiendo.....	13
3.4. Implementación	15
3.5. Ejemplos	17
4. Analizador semántico.....	20
4.1. Tabla de símbolos e implementación.....	20
4.2. Restricciones semánticas e implementación.....	21
4.2.1. Restricciones semánticas sobre la gramática.....	21
4.2.2. Restricciones semánticas sobre el AST	23
4.3. Construcción del árbol de Sintaxis Abstracta (AST) e implementación.....	24
4.3.1. Nodos del AST e implementación.....	24
4.3.2. Construcción del AST e implementación	26
4.4. Ejemplos	29
5. Conclusiones.....	32

Índice de ilustraciones

Ilustración 1. Máquina Discriminadora Determinista (MDD) asociada.	6
Ilustración 2. Análisis léxico del programa pruebaProfesor.	8
Ilustración 3. Análisis sintáctico del programa pruebaProfesor.	18
Ilustración 4. Análisis sintáctico del programa pruebaProfesor.	18
Ilustración 5. Análisis sintáctico del programa pruebaErrorSintactico.	19
Ilustración 6. Análisis semántico del programa pruebaProfesor.	30
Ilustración 7. Análisis semántico del programa pruebaCorrecto.	31
Ilustración 8. Análisis semántico del programa pruebaErrorSemantico.	32

Índice de tablas

Tabla 1. Categorías léxicas del lenguaje.	5
Tabla 2. Nodos del AST.	25

1. Introducción

Los compiladores son programas con los que trabajamos a menudo, usándolos como herramientas para crear programas. Aun así, solemos desconocer el funcionamiento interno de estos. Por tanto, el objetivo de esta práctica es **implementar un compilador** (sin incluir la generación de código intermedio) de un lenguaje simple, para comprender mejor su funcionamiento y la problemática asociada.

Para esto se implementarán los tres niveles de la fase de análisis: **léxico, sintáctico y semántico**. La implementación se realizará en **Python 3**, utilizando y aplicando los conocimientos adquiridos en las clases de teoría.

Para la realización de la práctica, la distribución del trabajo entre los integrantes ha sido equitativa: ambos han colaborado en la implementación de todos los analizadores, evitando “repartir” el trabajo y asignar a cada miembro un analizador concreto. De esta forma, ambos integrantes podrán adquirir un conocimiento más profundo sobre el funcionamiento y la implementación.

2. Analizador léxico

En este apartado se discutirá el desarrollo del **analizador léxico** utilizado por el compilador a implementar.

Concretamente se discutirán la especificación formal del lenguaje, la Máquina Discriminadora Determinista (MDD) asociada y una descripción de como se ha realizado la implementación. Finalmente, se mostrarán algunos ejemplos del funcionamiento correcto del analizador.

2.1. Especificación formal del lenguaje

A continuación, vamos a ver la especificación del lenguaje (descrita en la **Tabla 1**), viendo las categorías que lo componen.

Para cada categoría describiremos su expresión regular, qué acciones realizar al crearlas y los atributos que tienen. Todas las categorías tendrán un atributo adicional indicando la línea en la que se encuentran, que omitiremos aquí por simplicidad.

Es interesante comentar que las categorías de **Palabra reservada** e **Identificador** se solapan entre ellas (una palabra reservada podría identificarse como identificador). Para evitar esto, se respetará el orden establecido en la tabla (es decir, si un lexema se puede identificar tanto como palabra reservada como identificador se clasificará como palabra reservada). Esto se describirá posteriormente en el MDD.

Categoría	Expresión regular	Acciones	Atributos
Palabra reservada	PROGRAMA VAR VECTOR ENTERO REAL BOOLEANO INICIO FIN SI ENTONCES SINO MIENTRAS HACER LEE ESCRIBE Y O NO CIERTO FALSO	copiar palabra	palabra
Blanco	[<code>\t</code>]+	omitir	
Salto de línea	<code>\n</code>	omitir	
EOF	EOF	emitir	
Operador Asignación	<code>:=</code>	emitir	
Operador Suma	<code>[+]</code>	copiar operación	operación
Operador Multiplicación	<code>[*/]</code>	copiar operación	operación
Numero	<code>[0-9]+([0-9]+)?</code>	copiar tipo y valor	valor tipo
Identificador	<code>[a-zA-Z][a-zA-Z0-9]*</code>	copiar valor	valor
Operador Relacional	<code>= < > <= >= ></code>	copiar operación	operación
Paréntesis de Apertura	<code>(</code>	emitir	
Paréntesis de Cierre	<code>)</code>	emitir	
Corchete de Apertura	<code>[</code>	emitir	
Corchete de Cierre	<code>]</code>	emitir	
Punto	<code>.</code>	emitir	
Dos Puntos	<code>:</code>	emitir	
Punto y Coma	<code>;</code>	emitir	
Coma	<code>,</code>	emitir	

Tabla 1. Categorías léxicas del lenguaje.

Toda la información concreta sobre las categorías léxicas y su implementación se encuentra en el fichero *componentes.py*.

2.2. Máquina Discriminadora Determinista (MDD) asociada

Junto a la especificación del lenguaje, el compilador tiene una **Máquina Discriminadora Determinista (MDD)** asociada que se encarga de indicar cómo se segmenta la entrada (los caracteres leídos) en componentes léxicos.

La MDD asociada al compilador se puede observar en la **Ilustración 1**. Al tener la MDD un tamaño y complejidad considerables, y posiblemente no verse bien en el documento, se incluyen junto a la memoria los ficheros *mdd.png* y *mdd.jff* (formato de jFlap) para poder observar por separado la MDD.

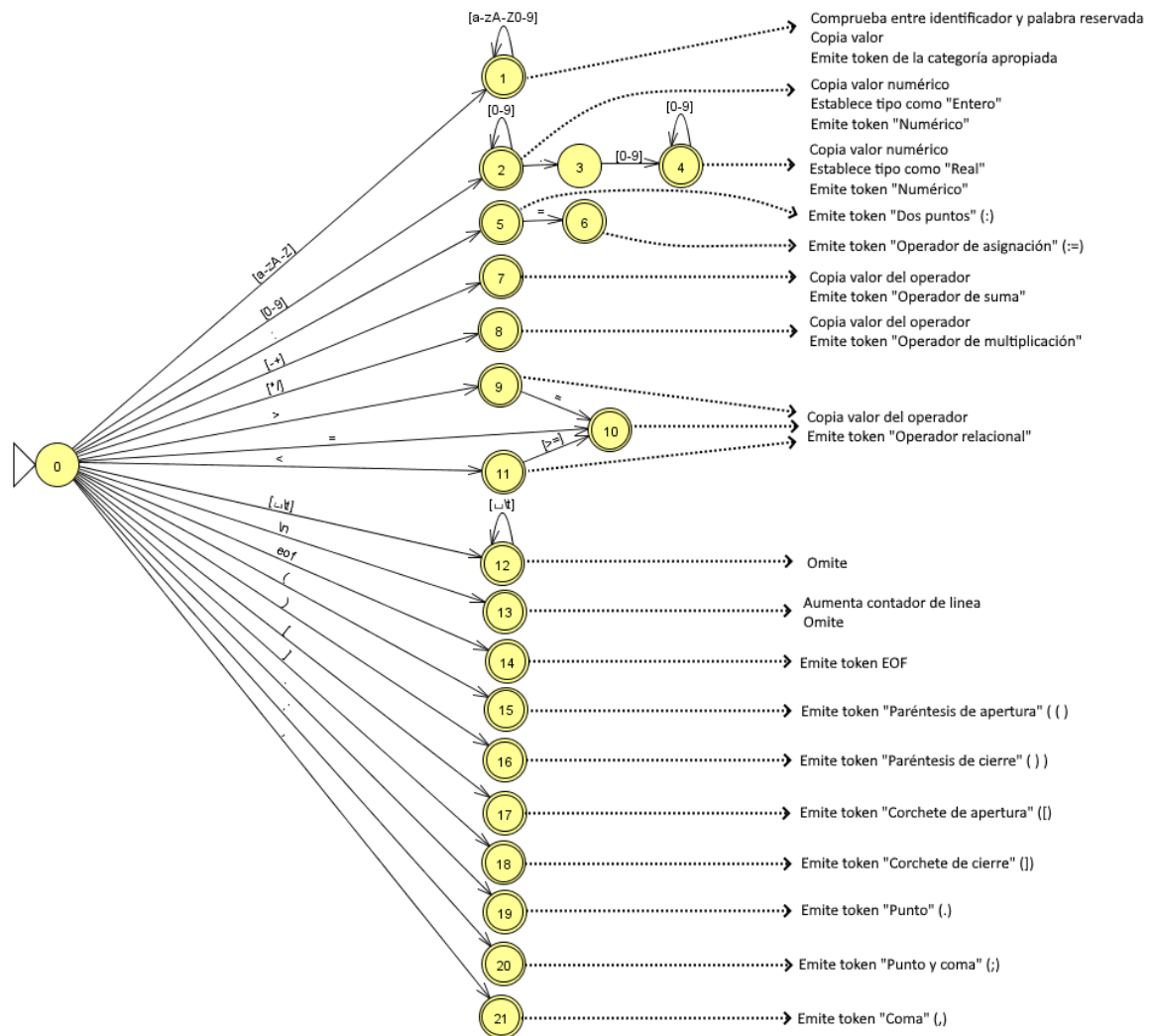


Ilustración 1. Máquina Discriminadora Determinista (MDD) asociada.

Se puede observar que no hay una rama específica para identificar palabras reservadas en la MDD. Para evitar complicar excesivamente la estructura, simplemente se hará la comprobación tras encontrar un identificador: si el identificador se corresponde con una palabra reservada, se emitirá esa palabra reservada en su lugar.

2.3. Implementación

Ahora vamos a ver como se ha realiza la **implementación** del analizador léxico.

Para iniciarla, se le pasará un fichero sobre el que se abrirá un flujo (para ir leyendo los caracteres del fichero). Tras esto se llamará al método **Analiza()** sobre el flujo anteriormente mencionado, lo que comienza el proceso de análisis léxico. Cada vez que llamemos al método Analiza, este nos devolverá el siguiente componente léxico que lea a partir del flujo de entrada. Para el funcionamiento del programa, se llamará en bucle a Analiza hasta que devuelva un EOF.

Dentro de este método lo que hace es implementar la MDD descrita en el apartado anterior mediante control de flujo (estructuras de tipo switch implementadas en Python utilizando *if/else*). El método irá leyendo caracteres y pasando de un sitio a otro conforme corresponda (dependiendo del carácter o caracteres que lleva leídos y del carácter que recibe), emitiendo finalmente el componente apropiado una vez sea necesario. Por ejemplo, al encontrar un "." emitirá un elemento (creará una instancia de la clase) de la categoría léxica Punto.

Debemos además especificar cómo esperamos que se unan las categorías para formar la entrada. Se va a implementar una "estrategia avariciosa", que consiste en agrupar el prefijo más largo que pertenezca a una categoría léxica, para que la entrada coincida con la que esperamos.

Se han realizado métodos aparte para el tratamiento de categorías léxicas recursivas: blancos, comentarios, identificadores y números:

- **TrataBlanco(flujo, ch)**: se encarga de omitir todos los blancos seguidos de golpe.
- **TrataComent(flujo, ch)**: se encarga de omitir los comentarios (desde `//` hasta `\n`, no se incluye el final de línea).
- **TrataIdent(flujo, ch)**: se llama cuando se encuentra un carácter alfabético y va leyendo hasta encontrar un carácter que no sea ni alfabético ni numérico. Este método devuelve el lexema del elemento leído (ya sea identificador o palabra reservada).
- **TrataNum(flujo, ch)**: de forma parecida al anterior, este método va leyendo números hasta que encuentra algo que no es un número.

Si lo que encuentra después es un punto, esperamos que el número sea real, por lo que seguimos leyendo números una vez más. Para indicar el tipo (entero o real) se usa una variable local *real*, que se pone en verdadero si hemos encontrado un punto y un dígito más. El método devuelve el número correspondiente (como entero o real dependiendo del caso).

Para diferenciar identificadores de palabras reservadas, una vez que hemos llamado a **TrataIdent(flujo, ch)** comprobamos si lo que nos devuelve está en el conjunto de palabras reservadas (están todas incluidas en un *FrozenSet*). Si lo está emitimos la palabra reservada correspondiente, y si no lo está, un identificador.

Un elemento a destacar del tratamiento de componentes léxicos es el tratamiento de los saltos de línea. Primero, hay que mencionar que se consideran tanto los caracteres `\r` (retorno de carro) como `\n` (línea nueva) para evitar errores dependientes de la codificación. Además, cada vez que se lee un salto de línea se incrementará una variable **nlinea**, usada para indicar la línea en la que se encuentra cada componente léxico.

Finalmente, si se ha llamado directamente al analizador léxico (en vez de al analizador sintáctico), se imprimirán por pantalla todos los componentes léxicos identificados.

Toda la implementación del analizador léxico se encuentra en el fichero *analex.py*.

2.4. Ejemplos

Para demostrar el correcto funcionamiento del analizador léxico, se mostrará la salida de este con un programa de ejemplo.

El **ejemplo** se corresponde con una versión adaptada del programa que se nos ha suministrado como ejemplo. El programa (contenido en el fichero *pruebaProfesor.txt*) es el siguiente:

```
PROGRAMA p1; //COMENTARIO
VAR i,x:ENTERO;
INICIO
    i:=0;
    MIENTRAS (i<>5) HACER
        INICIO
            x:=i*4;
            ESCRIBE(x);
        FIN; //COMENTARIO
FIN.
```

Como se puede observar en la ilustración, el funcionamiento del analizador léxico es el esperado: identifica todos los componentes del programa adecuadamente, indicando además la línea en la que se encuentran. Se puede ver también que se han ignorado los espacios en blanco y los comentarios adecuadamente.

```
Este es tu fichero 'pruebaProfesor.txt'
PalabraReservada (línea: 1, palabra: PROGRAMA)
Identificador (línea: 1, valor: p1)
PuntoComa (línea: 1)
PalabraReservada (línea: 2, palabra: VAR)
Identificador (línea: 2, valor: i)
Coma (línea: 2)
Identificador (línea: 2, valor: x)
DosPuntos (línea: 2)
PalabraReservada (línea: 2, palabra: ENTERO)
PuntoComa (línea: 2)
PalabraReservada (línea: 3, palabra: INICIO)
Identificador (línea: 4, valor: i)
OpAsigna (línea: 4)
Numero (línea: 4, numero: 0, tipo: int)
PuntoComa (línea: 4)
PalabraReservada (línea: 5, palabra: MIENTRAS)
ParentesisApertura (línea: 5)
Identificador (línea: 5, valor: i)
OpRelacional (línea: 5, operacion: <>)
Numero (línea: 5, numero: 5, tipo: int)
ParentesisCierre (línea: 5)
PalabraReservada (línea: 5, palabra: HACER)
PalabraReservada (línea: 6, palabra: INICIO)
Identificador (línea: 7, valor: x)
OpAsigna (línea: 7)
Identificador (línea: 7, valor: i)
OpMultiplicacion (línea: 7, operacion: *)
Numero (línea: 7, numero: 4, tipo: int)
PuntoComa (línea: 7)
PalabraReservada (línea: 8, palabra: ESCRIBE)
ParentesisApertura (línea: 8)
Identificador (línea: 8, valor: x)
ParentesisCierre (línea: 8)
PuntoComa (línea: 8)
PalabraReservada (línea: 9, palabra: FIN)
PuntoComa (línea: 9)
PalabraReservada (línea: 10, palabra: FIN)
Punto (línea: 10)
```

Ilustración 2. Análisis léxico del programa *pruebaProfesor*.

3. Analizador sintáctico

En este apartado se discutirá el desarrollo del **analizador sintáctico** utilizado en el compilador a implementar.

Concretamente se discutirán las producciones del lenguaje usadas, la transformación de la gramática a LL(1) en caso de ser necesario, los primeros y siguientes que podemos extraer de las producciones/no terminales y una descripción de como se ha realizado la implementación. Finalmente, se mostrarán algunos ejemplos del funcionamiento correcto del analizador sintáctico.

3.1. Producciones del lenguaje

A continuación, se mostrarán todas las producciones (sin modificar) del lenguaje:

1. **<Programa>** → PROGRAMA id ; <decl_var> <instrucciones> .
2. **<decl_var>** → VAR <lista_id> : <tipo> ; <decl_v>
3. **<decl_var>** → λ
4. **<decl_v>** → <lista_id> : <tipo> ; <decl_v>
5. **<decl_v>** → λ
6. **<lista_id>** → id <resto_listaid>
7. **<resto_listaid>** → , <lista_id>
8. **<resto_listaid>** → λ
9. **<Tipo>** → <tipo_std>
10. **<Tipo>** → VECTOR [num] DE <Tipo_std>
11. **<Tipo_std>** → ENTERO
12. **<Tipo_std>** → REAL
13. **<Tipo_std>** → BOOLEANO
14. **<instrucciones>** → INICIO <lista_inst> FIN
15. **<lista_inst>** → <instrucción> ; <lista_inst>
16. **<lista_inst>** → λ
17. **<instrucción>** → INICIO <lista_inst> FIN
18. **<instrucción>** → <inst_simple>
19. **<instrucción>** → <inst_es>
20. **<instrucción>** → SI <expresión> ENTONCES <instrucción> SINO <instrucción>
21. **<instrucción>** → MIENTRAS <expresión> HACER <instrucción>
22. **<inst_simple>** → id <resto_instsimple>
23. **<resto_instsimple>** → opasigna <expresión>
24. **<resto_instsimple>** → [<expr_simple>] opasigna <expresión>
25. **<resto_instsimple>** → λ
26. **<variable>** → id <resto_var>
27. **<resto_var>** → [<expr_simple>]
28. **<resto_var>** → λ
29. **<inst_es>** → LEE (id)
30. **<inst_es>** → ESCRIBE (<expr_simple>)
31. **<expresión>** → <expr_simple>
32. **<expresión>** → <expr_simple> oprel <expr_simple>
33. **<expr_simple>** → <término> <resto_exsimple>
34. **<expr_simple>** → <signo> <término> <resto_exsimple>
35. **<resto_exsimple>** → opsuma <término> <resto_exsimple>
36. **<resto_exsimple>** → O <término> <resto_exsimple>

- 37. **<resto_exsimple>** → λ
- 38. **<término>** → **<factor>** **<resto_term>**
- 39. **<resto_term>** → opmult **<factor>** **<resto_term>**
- 40. **<resto_term>** → Y **<factor>** **<resto_term>**
- 41. **<resto_term>** → λ
- 42. **<factor>** → **<variable>**
- 43. **<factor>** → num
- 44. **<factor>** → (**<expresión>**)
- 45. **<factor>** → NO **<factor>**
- 46. **<factor>** → CIERTO
- 47. **<factor>** → FALSO
- 48. **<signo>** → +
- 49. **<signo>** → -

3.2. Comprobación y transformación de la gramática a LL(1)

En este apartado vamos a comprobar si la gramática del punto anterior es LL(1) o no, para ello hemos de comprobar que no haya **recursión por la izquierda** y que no hayan **prefijos comunes**.

1. **Recursividad por la izquierda**: Se puede observar que ninguna producción de la gramática tiene recursividad por la izquierda.
2. **Prefijos comunes**: Se pueden observar prefijos comunes en las siguientes producciones de la gramática:

- 31. **<expresión>** → **<expr_simple>**
- 32. **<expresión>** → **<expr_simple>** oprel **<expr_simple>**

Procedemos a transformarlas para eliminar los prefijos comunes:

- 31. **<expresión>** → **<expr_simple>** **<expresiónPrime>**
- 32. **<expresiónPrime>** → oprel **<expr_simple>**
- 33. **<expresiónPrime>** → λ

Como la gramática transformada ya no contiene ni recursividad por la izquierda ni prefijos comunes, se puede afirmar que la nueva gramática es LL(1).

3.3. Primeros y siguientes de la gramática

En este apartado se calcularán los **primeros** y los **siguientes** de cada no terminal de la producción. Estos serán útiles para realizar la tabla de análisis (no realizada en esta memoria) y para la implementación, descrita posteriormente.

3.3.1. Primeros

A continuación, vamos a obtener el conjunto de primeros de cada regla e indicar si son anulables:

1. **<Programa>** → PROGRAMA id ; **<decl_var>** **<instrucciones>** .
{PROGRAMA}

2. **<decl_var>** → VAR <lista_id> : <tipo> ; <decl_v>
{VAR}
3. **<decl_var>** → λ
{ } **ANULABLE**
4. **<decl_v>** → <lista_id> : <tipo> ; <decl_v>
{id}
5. **<decl_v>** → λ
{ } **ANULABLE**
6. **<lista_id>** → id <resto_listaid>
{id}
7. **<resto_listaid>** → , <lista_id>
{ , }
8. **<resto_listaid>** → λ
{ } **ANULABLE**
9. **<Tipo>** → <tipo_std>
{ENTERO, REAL, BOOLEANO}
10. **<Tipo>** → VECTOR [num] DE <Tipo_std>
{VECTOR}
11. **<Tipo_std>** → ENTERO
{ENTERO}
12. **<Tipo_std>** → REAL
{REAL}
13. **<Tipo_std>** → BOOLEANO
{BOOLEANO}
14. **<instrucciones>** → INICIO <lista_inst> FIN
{INICIO}
15. **<lista_inst>** → <instrucción> ; <lista_inst>
{id, INICIO, LEE, ESCRIBE, SI, MIENTRAS}
16. **<lista_inst>** → λ
{ } **ANULABLE**
17. **<instrucción>** → INICIO <lista_inst> FIN
{INICIO}
18. **<instrucción>** → <inst_simple>
{id}
19. **<instrucción>** → <inst_es>
{LEE, ESCRIBE}
20. **<instrucción>** → SI <expresion> ENTONCES <instrucción> SINO <instrucción>
{SI}
21. **<instrucción>** → MIENTRAS <expresión> HACER <instrucción>
{MIENTRAS}
22. **<inst_simple>** → id <resto_instsimple>
{id}
23. **<resto_instsimple>** → opasigna <expresión>
{opasigna}
24. **<resto_instsimple>** → [<expr_simple>] opasigna <expresión>
{ [] }
25. **<resto_instsimple>** → λ
{ } **ANULABLE**
26. **<variable>** → id <resto_var>
{id}
27. **<resto_var>** → [<expr_simple>]
{ [] }

28. **<resto_var>** → λ
{} **ANULABLE**
29. **<inst_es>** → LEE (id)
{LEE}
30. **<inst_es>** → ESCRIBE (<expr_simple>)
{ESCRIBE}
31. **<expresión>** → <expr_simple> <expresiónPrime>
{id, num, (, NO, CIERTO, FALSO, +, -}
32. **<expresiónPrime>** → oprel <expr_simple>
{oprel}
33. **<expresiónPrime>** → λ
{} **ANULABLE**
34. **<expr_simple>** → <término> <resto_exsimple>
{id, num, (, NO, CIERTO, FALSO}
35. **<expr_simple>** → <signo> <término> <resto_exsimple>
{+, -}
36. **<resto_exsimple>** → opsuma <término> <resto_exsimple>
{opsuma}
37. **<resto_exsimple>** → O <término> <resto_exsimple>
{O}
38. **<resto_exsimple>** → λ
{} **ANULABLE**
39. **<término>** → <factor> <resto_term>
{id, num, (, NO, CIERTO, FALSO}
40. **<resto_term>** → opmult <factor> <resto_term>
{opmult}
41. **<resto_term>** → Y <factor> <resto_term>
{Y}
42. **<resto_term>** → λ
{} **ANULABLE**
43. **<factor>** → <variable>
{id}
44. **<factor>** → num
{num}
45. **<factor>** → (<expresión>)
{ (}
46. **<factor>** → NO <factor>
{NO}
47. **<factor>** → CIERTO
{CIERTO}
48. **<factor>** → FALSO
{FALSO}
49. **<signo>** → +
{+}
50. **<signo>** → -
{-}

3.3.2. Siguietes

Ahora procedemos a obtener el conjunto de siguientes de los no terminales de la gramática, que nos serán útiles en el análisis de no terminales anulables y para la sincronización en caso de errores:

- **Primer paso:** añadimos el final de fichero al no terminal inicial (en este caso, **Programa**):

<Programa>	\$
<decl_var>	
<decl_v>	
<lista_id>	
<resto_listaid>	
<Tipo>	
<Tipo_std>	
<instrucciones>	
<lista_inst>	
<instrucción>	
<inst_simple>	
<resto_instsimple>	
<variable>	
<resto_var>	
<inst_es>	
<expresión>	
<expresiónPrime>	
<expr_simple>	
<resto_exsimple>	
<término>	
<resto_term>	
<factor>	
<signo>	

- **Segundo paso:** Si $\langle A \rangle \rightarrow \alpha\beta$: $\text{primeros}(\beta) \subseteq \text{siguietes}(\rho)$

<Programa>	\$
<decl_var>	INICIO
<decl_v>	
<lista_id>	:
<resto_listaid>	
<Tipo>	;
<Tipo_std>	
<instrucciones>	.
<lista_inst>	FIN
<instrucción>	;, SINO
<inst_simple>	
<resto_instsimple>	
<variable>	
<resto_var>	
<inst_es>	
<expresión>	HACER,), ENTONCES
<expresiónPrime>	

<expr_simple>],), oprel
<resto_exsimple>	
<término>	opsuma, O
<resto_term>	
<factor>	opmult, Y
<signo>	id, num, (, NO, CIERTO, FALSO

- **Tercer paso:** Mientras haya cambios, repetir:
Si $\langle A \rangle \rightarrow \alpha\beta$ y β es anulable: $\text{siguientes}(\langle A \rangle) \subseteq \text{siguientes}(\rho)$

Primera iteración:

<Programa>	\$
<decl_var>	INICIO
<decl_v>	INICIO
<lista_id>	:
<resto_listaid>	:
<Tipo>	;
<Tipo_std>	;
<instrucciones>	.
<lista_inst>	FIN
<instrucción>	;; SINO
<inst_simple>	;; SINO
<resto_instsimple>	;; SINO
<variable>	opmult, Y
<resto_var>	opmult, Y
<inst_es>	;; SINO
<expresión>	HACER,), ENTONCES, ;; SINO
<expresiónPrime>	HACER,), ENTONCES, ;; SINO
<expr_simple>],), oprel, HACER, ENTONCES, ;; SINO
<resto_exsimple>],), oprel, HACER, ENTONCES, ;; SINO
<término>	opsuma, O,],), oprel, HACER, ENTONCES, ;; SINO
<resto_term>	opsuma, O,],), oprel, HACER, ENTONCES, ;; SINO
<factor>	opmult, Y, opsuma, O,],), oprel, HACER, ENTONCES, ;; SINO
<signo>	id, num, (, NO, CIERTO, FALSO

Segunda iteración:

<Programa>	\$,
<decl_var>	INICIO
<decl_v>	INICIO
<lista_id>	:
<resto_listaid>	:
<Tipo>	;
<Tipo_std>	;
<instrucciones>	.
<lista_inst>	FIN
<instrucción>	;; SINO
<inst_simple>	;; SINO
<resto_instsimple>	;; SINO

<variable>	opmult, Y, opsuma, O,],), oprel, HACER, ENTONCES, ;;, SINO
<resto_var>	opmult, Y, opsuma, O,],), oprel, HACER, ENTONCES, ;;, SINO
<inst_es>	;;, SINO
<expresión>	HACER,), ENTONCES, ;;, SINO
<expresiónPrime>	HACER,), ENTONCES, ;;, SINO
<expr_simple>],), oprel, HACER, ENTONCES, ;;, SINO
<resto_exsimple>],), oprel, HACER, ENTONCES, ;;, SINO
<término>	opsuma, O,],), oprel, HACER, ENTONCES, ;;, SINO
<resto_term>	opsuma, O,],), oprel, HACER, ENTONCES, ;;, SINO
<factor>	opmult, Y, opsuma, O,],), oprel, HACER, ENTONCES, ;;, SINO
<signo>	id, num, (, NO, CIERTO, FALSO

En la tercera iteración no encontramos ningún cambio, por lo que paramos aquí el proceso de cálculo de siguientes.

3.4. Implementación

Antes de nada, comentamos algunos métodos contenidos en el fichero *anasint.py* que se han usado para la implementación:

- **avanza():** lee el siguiente componente léxico.
- **<NombreDelNoTerminal>():** Analiza el no terminal que coincide con el nombre del método. Estos métodos contienen parámetros que no se especifican aquí (al ser parte del análisis semántico)
- **Error(num, linea, **opcional):** imprime un mensaje con el error correspondiente. Además, pone a falso la variable *aceptación*, que indica posteriormente que el análisis es fallido (ya que contiene errores el programa).

El valor de *num* determina cuál es el error que debe imprimirse, *linea* representa la línea donde se ha producido ese error (utilizada en el mensaje impreso) y *opcional* son valores que pueden ser utilizados para enriquecer algunos mensajes de error concretos (principalmente en el analizador semántico).

- **Sincroniza(categoríasSiguietes, reservadasSiguietes, categoria, reservada):** Método encargado de realizar la sincronización en modo pánico (que se describirá posteriormente). Los parámetros del método son los siguientes:
 - **CategoríasSiguietes:** categorías léxicas con las que se puede sincronizar.
 - **ReservadasSiguietes:** palabras reservadas con las que se puede sincronizar.
 - **Categoría:** categoría léxica concreta que se espera. Usado en casos que no se ha encontrado un terminal concreto, esperamos el terminal fallido. Si el valor es **PalabraReservada**, el siguiente parámetro se usa.
 - **Reservada:** Palabra reservada concreta que se espera. Si no se espera ninguna, su valor es None.

Este método avanza componentes léxicos hasta encontrar uno de los que le hemos pasado en los parámetros. Si se trata de un elemento “Siguiete” se queda en el último componente leído, mientras que si se trata de **categoría o reservada** avanzará una posición más antes de volver.

Para la **implementación**, se utilizará un **analizador descendente recursivo**. Concretamente, hemos seguido la siguiente estrategia:

1. Se llama al método para analizar el no terminal correspondiente.

Ejemplo: Al principio del análisis se llama a "**Programa()**" para analizar el no terminal inicial.

2. Se revisan los primeros de las reglas del no terminal.

Si el componente actual coincide con alguno de los primeros de alguna regla del no terminal, se elegirá esa regla y se pasa a analizarla.

Si no coincide con ninguno de los primeros pero el no terminal es anulable, comprobamos si el componente coincide con algunos de los siguientes del no terminal. Si lo hace saldremos del método sin hacer nada (equivalente a la producción palabra vacía).

En caso de que no sea anulable o que el componente no coincida con uno de los siguientes se proporcionará un mensaje de error con **Error(num, linea)**.

Ejemplo: En el no terminal *inst_es* se busca "LEE" o "ESCRIBE". Si se encuentra "ESCRIBE", se pasará a analizar la regla *<inst_es> → ESCRIBE (<expr_simple>)*.

3. Una vez que hemos entrado en una regla (hemos encontrado alguno de los primeros), pasamos a analizarla.

Si el primer elemento de la regla era un terminal, se llama a **avanza()** (pues ya se ha comprobado que dicho elemento está al comprobar los primeros de la regla). En caso contrario se llama al método de análisis del no terminal correspondiente.

A partir de esto, cada vez que se espera un no terminal se llama al método correspondiente, esperando a que termine para proseguir con el análisis. Si el siguiente elemento que esperamos es un terminal, se comprueba que es ese terminal y se llama a **avanza()** en caso de éxito, siguiendo con el análisis.

Si no se puede encontrar ese terminal, se lanza un mensaje de error con **Error(num, linea)**. No se lanzan mensajes de error cuando llamamos al método de un no terminal (ya que el propio no terminal se encargará de lanzar sus errores)

Ejemplo: Una vez que hemos entrado en *<inst_es> → ESCRIBE (<expr_simple>)* se llama a **avanza()** ya que hemos encontrado "ESCRIBE". Tras eso buscamos el siguiente elemento, "("". Cuando lo leemos llamamos de nuevo a **avanza()**. Ahora nos encontramos ante el no terminal *expr_simple*, por lo que llamamos a **expr_simple()** para que analice su parte. Una vez nos devuelva el control terminaremos de leer el último ")", llamaremos a **avanza()** y volveremos al método que ha llamado a este método.

El método descrito anteriormente es el flujo normal de análisis, pero como ya hemos adelantado, **podemos tener errores**. Vamos a ver como se ha implementado el **modo pánico** para seguir analizando en caso de error, pues queremos encontrar la mayor cantidad de errores posibles en una sola compilación:

- Cuando se produce un error en el 2º paso (esto es, que no puede entrar a ninguna regla porque no coincide con ningún primero o siguientes en caso de ser anulable), **sincronizamos** hasta encontrar un primero (volviendo a iniciar el análisis del no terminal) o encontrar un siguiente (omitiendo el análisis de este no terminal y devolviendo el control a la función que llamó al método)

Ejemplo: En el no terminal `inst_es` se busca "LEE" o "ESCRIBE", pero no se encuentra ninguno. Se va avanzando hasta encontrar "LEE", "ESCRIBE" o un siguiente del no terminal (";" o "SINO"). Si por ejemplo encontramos "ESCRIBE", pasaremos a analizar la regla `<inst_es> → ESCRIBE (<expr_simple>)`, pero si encontramos ";", haremos un return.

- Si el error nos lo encontramos en el 3er paso (esto es, que no puede encontrar el terminal que espera leer), **sincronizamos** hasta encontrar el terminal que corresponda a los siguientes del propio no terminal, a los siguientes del elemento que esperaba encontrar o al elemento que se esperaba encontrar.

Si se obtiene el elemento que se esperaba o un siguiente al elemento que se esperaba encontrar se continúa el análisis a partir de ese punto (avanzando un componente extra en el caso de encontrar el terminal esperado), en caso contrario (siguientes del no terminal) se devuelve el control y aborta el análisis de ese no terminal.

Ejemplo: En `<inst_es> → ESCRIBE (<expr_simple>)` buscamos el elemento "(", pero no lo encontramos. Ahora buscamos el propio elemento, el elemento siguiente a él (es decir, los primeros de `expr_simple` (un identificador)) o los siguientes de `inst_es` (";" o "SINO"). Si encontramos un identificador o el paréntesis de apertura, continuamos con el análisis desde el punto en el que nos habíamos quedado (`expr_simple()`). Si se encuentra en su lugar un ";", haríamos un return y continuaríamos con el análisis.

En el caso en el que en una sincronización se encuentre el final de fichero "EOF" antes de lo esperado, se imprimirá un error especial de final de fichero inesperado y se detendrá el análisis de errores.

Toda la implementación del analizador sintáctico (junto a la del analizador semántico) se encuentra detallada en el fichero *anasint.py*.

3.5. Ejemplos

Para demostrar el correcto funcionamiento del analizador sintáctico, mostraremos los resultados de la ejecución de varios programas de prueba.

El **primer ejemplo** se corresponde con una versión adaptada del programa que se nos ha suministrado como ejemplo. El programa (contenido en el fichero *pruebaProfesor.txt*) es el siguiente:

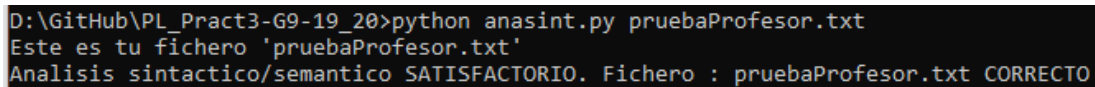
```
PROGRAMA p1; //COMENTARIO
VAR i,x:ENTERO;
```

```

INICIO
    i:=0;
    MIENTRAS (i<>5) HACER
        INICIO
            x:=i*4;
            ESCRIBE(x);
        FIN; //COMENTARIO
FIN.

```

La salida del analizador se puede observar en la siguiente figura. El programa se analiza sin encontrar ningún error y el AST obtenido es coherente y representa adecuadamente a la entrada.



```

D:\GitHub\PL_Pract3-G9-19_20>python anasint.py pruebaProfesor.txt
Este es tu fichero 'pruebaProfesor.txt'
Análisis sintactico/semantico SATISFACTORIO. Fichero : pruebaProfesor.txt CORRECTO

```

Ilustración 3. Análisis sintáctico del programa pruebaProfesor.

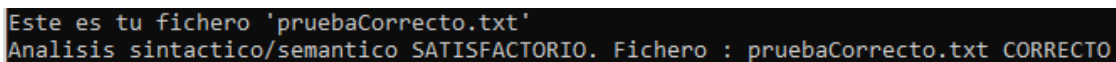
El **segundo ejemplo** se corresponde con un programa distinto, creado para intentar probar estructuras más complejas. El programa (contenido en el fichero *pruebaCorrecto.txt*) es el siguiente:

```

PROGRAMA p1;
VAR a : VECTOR [5] DE BOOLEANO;
    b, c, d: REAL;
INICIO
    a[1] := 1.1 < 2.2;
    MIENTRAS a[2] HACER b := (c*d) + 2;
    SI a[3] O a[4] ENTONCES
        c := -1.1
    SINO
        d := 0.0;
FIN.//COMENTARIO DE PRUEBA

```

La salida del analizador se puede observar en la siguiente figura. De nuevo, el programa se analiza sin encontrar ningún error y el AST se crea correctamente y de forma coherente.



```

Este es tu fichero 'pruebaCorrecto.txt'
Análisis sintactico/semantico SATISFACTORIO. Fichero : pruebaCorrecto.txt CORRECTO

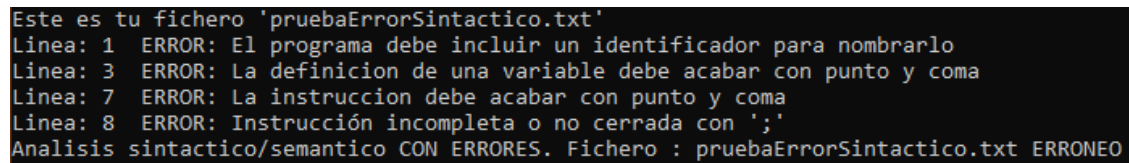
```

Ilustración 4. Análisis sintáctico del programa pruebaProfesor.

El **último** ejemplo se corresponde con un programa que contiene algunos errores sintácticos, buscando comprobar el funcionamiento adecuado de la sincronización en modo pánico. El programa (contenido en el fichero *pruebaErrorSintáctico.txt*) es el siguiente:

```
PROGRAMA ; //COMENTARIO DE PRUEBA
VAR a: ENTERO;
    b: REAL
INICIO
    MIENTRAS 1 <> 2 HACER
        a := 2;
    LEE(a)
    a := 1+2
    a := 1+3;
FIN.
```

Se puede ver que el programa contiene numerosos errores semánticos (falta el identificador del programa y hay varias instrucciones sin punto y coma). Como se puede observar en la ilustración, gracias a la sincronización se han podido detectar todos los errores en una única ejecución.



```
Este es tu fichero 'pruebaErrorSintactico.txt'
Linea: 1  ERROR: El programa debe incluir un identificador para nombrarlo
Linea: 3  ERROR: La definicion de una variable debe acabar con punto y coma
Linea: 7  ERROR: La instruccion debe acabar con punto y coma
Linea: 8  ERROR: Instrucción incompleta o no cerrada con ';'
Análisis sintactico/semantico CON ERRORES. Fichero : pruebaErrorSintactico.txt ERRONEO
```

Ilustración 5. Análisis sintáctico del programa pruebaErrorSintactico.

4. Analizador semántico

En este apartado se discutirá el desarrollo del **analizador semántico** utilizado en el compilador a implementar. Concretamente se discutirá la implementación usada para la tabla de símbolos, las restricciones semánticas (qué restricciones semánticas hemos incluido y cómo las hemos tratado) y la construcción del árbol de sintaxis abstracta.

Es importante destacar que, para implementar el analizador semántico, utilizaremos un **esquema de traducción dirigido por sintaxis**. Esto significa que las acciones semánticas y de construcción del árbol se encontrarán intercaladas en las producciones de la gramática del analizador sintáctico.

4.1. Tabla de símbolos e implementación

La **tabla de símbolos** es una estructura de datos que contiene información relativa a los identificadores contenidos en el programa. Esta información será usada posteriormente para la comprobación de las restricciones semánticas (también podría ser usada durante la generación de código, aunque esa parte no se incluirá en el desarrollo de esta práctica)

Concretamente, la información que contendrá nuestra tabla de símbolos será:

- **Identificador:** nombre del identificador.
- **Tipo:** tipo de la información que contiene el identificador. Puede ser *entero*, *real*, *booleano* o *programa* (para simbolizar el identificador del programa)
- **Clase:** clase de variable asociada al identificador. Puede ser *variable* (para variables simples), *vector* o *programa* (para simbolizar el identificador del programa)
- **Longitud:** utilizada únicamente para identificadores de clase vector. Contiene el tamaño del vector (entendiendo como tamaño la cantidad de elementos que contiene)

La razón para separar *tipo* y *clase* es permitir distinguir de forma sencilla variables sencillas de vectores pudiendo conocer su tipo de cara a comprobaciones semánticas.

La **implementación** utilizada para la tabla de símbolos ha sido un **diccionario de diccionarios**. Concretamente, el *identificador* será la clave del diccionario, y con esta clave podremos acceder a otro diccionario (un diccionario para cada identificador) que contiene el resto de información sobre el identificador (siendo *tipo*, *clase* y *longitud* las claves para acceder a la información correspondiente)

Además de este diccionario de diccionarios, se incluyen tres métodos para manipular la tabla de símbolos de forma controlada:

- **anadeSimbolo(símbolo, tipo):** Añade el símbolo indicado a la tabla de símbolos con el tipo indicado como parámetro. En caso de éxito devolverá True, pero en caso de fallo puede devolver uno de dos errores:
 - Si el símbolo que intentamos añadir a la tabla no es válido (es equivalente a una palabra reservada) devolverá un error **Invalido**.
 - Si el símbolo que intentamos añadir a la tabla ya está contenido en la tabla, se devolverá un error **Duplicado**.

- **actualizaInfo(símbolo, nombreInfo, valorInfo)**: Añade al símbolo especificado un valor que se corresponde con el nombre indicado. Por ejemplo, *actualizaInfo(a, "tipo", "entero")* añadirá al diccionario del símbolo *a* que su tipo es entero. En caso de éxito devolverá True, y en caso de error (el símbolo no está contenido en la tabla) devolverá False.
- **devuelveInfo(símbolo, nombreInfo)**: Recupera una información concreta asociada a un símbolo. Por ejemplo, *devuelveInfo(a, "tipo")* devolverá el tipo del símbolo *a*. En caso de éxito devolverá el valor, y en caso de error devolverá None.

El tratamiento de los errores de estas funciones se comentará en el próximo apartado.

Tanto el diccionario de diccionarios como los tres métodos para manipularlo han sido implementados en un fichero aparte (*tablaSimbolos.py*) para facilitar el acceso por parte del analizador sintáctico/semántico y de los nodos del AST.

4.2. Restricciones semánticas e implementación

Las restricciones semánticas que se han tenido en cuenta han sido diversas, yendo desde la comprobación de que no se declaran varias veces variables con el mismo identificador hasta comprobar que los tipos de las operaciones sean apropiados.

Hay que mencionar que, igual que se considera que el análisis ha sido fallido si no se respetan las reglas de la gramática en el análisis sintáctico, **cualquier violación de las restricciones semánticas** provocará que el análisis se considere **fallido**, imprimiendo los mensajes apropiados por pantalla.

Las restricciones han sido implementadas de dos formas distintas: parte de ellas han sido comprobadas en **la gramática**, y parte han sido comprobadas en el **árbol de sintaxis abstracta**. Ambos tipos de restricciones se describirán por separado:

4.2.1. Restricciones semánticas sobre la gramática

Las restricciones semánticas que se han implementado sobre la gramática tratan principalmente sobre la **declaración correcta de variables**. Concretamente, las restricciones que se comprueban son las siguientes:

- **El símbolo no ha sido declarado previamente**: cada símbolo puede ser declarado una única vez, no se permite la repetición de símbolos.
- **Los símbolos no pueden compartir lexema con las palabras reservadas**: no se permite que un símbolo coincida con una palabra reservada del lenguaje. Por ejemplo, no se permiten los símbolos *"programa"* ni *"Programa"* (al coincidir estos con la palabra reservada PROGRAMA).
- **El tamaño de los vectores debe ser un número entero positivo**: los vectores no pueden tener tamaños negativos ni iguales a cero; y el tamaño del vector debe ser un número entero (no se permiten números reales).

Para la **implementación** de estas restricciones, se han añadido acciones intercaladas a las producciones del lenguaje. Concretamente, las producciones con las acciones intercaladas son:

1. **<Programa>** → PROGRAMA id
*{si equivaleAReservada(id.lexema) entonces error_simbolo_invalido
 sino si tablaSimbolos.contiene(id.lexema) entonces error_simbolo_repetido
 sino
 tablaSimbolos.anadeSimbolo(id.lexema, "programa")
 tablaSimbolos.actualizaInfo(id.lexema, "clase", "programa")
 fin si};*
 <decl_var> <instrucciones> .
2. **<decl_var>** → VAR <lista_id> : <tipo> ;
*{para v en <lista_id>.lista hacer
 si equivaleAReservada(id.lexema) entonces error_simbolo_invalido
 sino si tablaSimbolos.contiene(id.lexema) entonces error_simbolo_repetido
 sino
 tablaSimbolos.anadeSimbolo(v, <tipo>.t)
 tablaSimbolos.actualizaInfo(v, "clase", <tipo>.clase)
 si <tipo>.clase == "vector" entonces tablaSimbolos.actualizaInfo(v, "longitud",
 <tipo>.longitud)
 fin si
 fin para} <decl_v>*
3. **<decl_var>** → λ
4. **<decl_v>** → <lista_id> : <tipo> ;
*{para v en <lista_id>.lista hacer
 si equivaleAReservada(id.lexema) entonces error_simbolo_invalido
 sino si tablaSimbolos.contiene(id.lexema) entonces error_simbolo_repetido
 sino
 tablaSimbolos.anadeSimbolo(v, <tipo>.t)
 tablaSimbolos.actualizaInfo(v, "clase", <tipo>.clase)
 si <tipo>.clase == "vector" entonces tablaSimbolos.actualizaInfo(v, "longitud",
 <tipo>.longitud)
 fin si
 fin para} <decl_v>*
5. **<decl_v>** → λ
6. **<lista_id>** → id {<resto_listaid>.lh = [id.lexema]} <resto_listaid>
{<lista_id>.lista = <resto_listaid>.lista}
7. **<resto_listaid>** → , <lista_id> {<resto_listaid>.lista = <resto_listaid>.lh ++ <lista_id>.lista}
8. **<resto_listaid>** → λ {<resto_listaid>.lista = <resto_listaid>.lh}
9. **<Tipo>** → <tipo_std> {<Tipo>.t = <tipo_std>.t; <Tipo>.clase = "variable", <Tipo>.longitud = 0}
10. **<Tipo>** → VECTOR [num] DE <Tipo_std> {<Tipo>.t = <tipo_std>.t; <Tipo>.clase = "vector"; <Tipo>.longitud = num.valor}
11. **<Tipo_std>** → ENTERO {<tipo_std>.t = "entero"}
12. **<Tipo_std>** → REAL {<tipo_std>.t = "real"}
13. **<Tipo_std>** → BOOLEANO {<tipo_std>.t = "booleano"}

Estas acciones se encargan de **introducir** los símbolos en la tabla de símbolos, y de comprobar las restricciones semánticas descritas. La implementación consiste en intercalar las acciones directamente en el analizador descendente recursivo, en la posición que les corresponde dentro de la producción. Cuando sea necesario, también se encargarán de imprimir los mensajes de error necesarios.

Para almacenar los atributos de los no terminales, se ha utilizado una clase **Atributos**, que no es más que un *wrapper* para un diccionario. Se pasa una instancia de esta clase como parámetro a cada llamada a los métodos del analizador para almacenar los atributos tanto heredados como sintetizados.

Todo el detalle de la implementación se encuentra en el fichero *anasint.py*.

4.2.2. Restricciones semánticas sobre el AST

El resto de las restricciones semánticas se comprueban directamente sobre los nodos del árbol de sintaxis abstracto, al resultar más fáciles de comprobar de esta forma. Las restricciones que se comprueban sobre el árbol en concreto son las siguientes:

- **Se realiza conversión implícita de entero a real cuando sea necesario:** Tanto en asignaciones (cuando asignamos un valor entero a una variable real) como en expresiones aritméticas (cuando un operador es real y otro entero), se transformará de forma implícita el valor entero a un valor real.
- **No se realiza conversión implícita de tipos booleanos:** las constantes booleanas pueden tomar únicamente dos valores: CIERTO o FALSO. Además, no hay conversión de valores booleanos a numéricos en ningún caso.
- **Los tipos en una asignación deben ser compatibles:** cuando asignamos un valor a una variable (ya sea simple o un vector), este valor debe ser compatible con la variable. Las variables **enteras** aceptan únicamente valores enteros, las variables **reales** aceptan valores numéricos (realizando conversión de entero a real si fuese necesario) y las variables **booleanas** solo aceptan valores booleanos.
- **Las condiciones deben ser de tipo lógico:** las condiciones de las estructuras de **MIENTRAS** y **SI ENTONCES SINO** deben ser de tipo lógico (tienen que ser evaluables como CIERTO o FALSO), no pueden ser numéricas.
- **La instrucción LEE solo admite variables simples numéricas:** la instrucción LEE solo admite variables simples (no acepta vectores) y numéricas (no acepta variables booleanas)
- **Las operaciones aritméticas, de comparación y de signo se deben realizar entre expresiones numéricas:** las operaciones aritméticas, de comparación y de signo se tienen que realizar entre expresiones numéricas, no se permiten expresiones lógicas.
- **Las operaciones lógicas deben realizarse entre expresiones lógicas:** de forma análoga a la restricción anterior, las operaciones lógicas (**Y**, **O** y **NO**) se deben realizar entre expresiones lógicas, no se admiten expresiones numéricas.
- **El acceso a las variables debe ser apropiado:** para acceder a una variable, esta debe de estar declarada previamente. Además, se tiene que respetar la clase de la variable (no se puede acceder a una posición del vector de una variable simple; y no se puede acceder directamente a un vector, teniendo que acceder a una posición del vector)

La **implementación** de estas restricciones se hace a través del método **compsem** implementado en cada nodo del AST. Este método se encarga de comprobar las restricciones correspondientes a cada nodo cuando éste se crea (accediendo a la tabla de símbolos si fuese necesario), almacenando en una lista (llamada *errores*) un código identificativo por cada error que se ha encontrado.

Esta lista se accederá en el analizador sintáctico con un método **comprobaciónSemánticaAST(nodo)**. Este método se llama inmediatamente después de crear cada nodo y se encarga de recorrer la lista de errores del nodo, imprimiendo por pantalla los errores apropiados y (si ha habido algún error) marcando el análisis como fallido.

Todo el detalle de la implementación de estas restricciones semánticas se encuentra en el fichero *AST.py*.

4.3. Construcción del árbol de Sintaxis Abstracta (AST) e implementación

Además de la comprobación de las restricciones semántica, la otra tarea del analizador semántico es la **construcción del árbol de sintaxis abstracta (AST)**, representando así el programa analizado de una forma que pueda ser utilizada por las fases posteriores de compilación o interpretación.

4.3.1. Nodos del AST e implementación

La primera tarea que realizaremos es especificar los tipos de nodos que pueden conformar el árbol de sintaxis abstracta. Estos quedan recogidos en la Tabla 1. Es interesante destacar que todos los nodos contienen dos atributos, **línea** y **profundidad**, utilizados para la impresión posterior del AST. También contienen un atributo **errores**, que fue comentado en el apartado anterior. Al ser atributos compartidos por todos los nodos, no quedarán recogidos en la tabla.

Tipo de nodo	Representa	Atributos	Hijos
Asignación	Sentencia de asignación; asignación de una expresión a un símbolo	---	<ul style="list-style-type: none"> • Símbolo al que se asigna • Expresión a asignar
Si	Sentencia condicional SI _ ENTONCES _ SINO _	---	<ul style="list-style-type: none"> • Condición • Instrucciones para el SI • Instrucciones para el SINO
Mientras	Sentencia de bucle MIENTRAS _ HACER _	---	<ul style="list-style-type: none"> • Condición • Instrucciones
Lee	Sentencia de lectura de una variable simple	<ul style="list-style-type: none"> • Variable simple a leer 	---
Escribe	Sentencia de escritura en pantalla	---	<ul style="list-style-type: none"> • Expresión a escribir
Compuesta	Sentencia compuesta; lista de instrucciones	---	<ul style="list-style-type: none"> • Lista de instrucciones
Comparación	Comparación de expresiones	<ul style="list-style-type: none"> • Tipo resultante • Operación realizada 	<ul style="list-style-type: none"> • Dos expresiones a comparar
Aritmético	Operación aritmética entre expresiones	<ul style="list-style-type: none"> • Tipo resultante • Operación realizada 	<ul style="list-style-type: none"> • Dos expresiones a operar
Entero	Valor de tipo entero	<ul style="list-style-type: none"> • Tipo resultante • Valor 	---

Real	Valor de tipo real	<ul style="list-style-type: none"> • Tipo resultante • Valor 	---
Booleano	Valor de tipo booleano	<ul style="list-style-type: none"> • Tipo resultante • Valor 	---
AccesoVariable	Acceso a una variable simple	<ul style="list-style-type: none"> • Tipo resultante • Identificador de la variable 	---
AccesoVector	Acceso a un vector	<ul style="list-style-type: none"> • Tipo resultante • Identificador del vector 	---
Signo	Signo aplicado a una expresión	<ul style="list-style-type: none"> • Tipo resultante • Signo utilizado 	<ul style="list-style-type: none"> • Expresión sobre la que aplicar el signo
Lógico	Operación lógica entre expresiones	<ul style="list-style-type: none"> • Tipo resultante • Operación realizada 	<ul style="list-style-type: none"> • Dos expresiones a operar
No	Operación lógica NO sobre una expresión	<ul style="list-style-type: none"> • Tipo resultante 	<ul style="list-style-type: none"> • Expresión sobre la que aplicar el NO lógico
Vacío	Nodo especial que simboliza error sintáctico o semántico durante la construcción del árbol	---	---

Tabla 2. Nodos del AST.

Aparte de los nodos iniciales que nos fueron indicados, se han añadido el nodo de **Signo** (para tratar con el no terminal *Signo*) y los nodos de **Lógico** y **No** (para tratar por separado las operaciones lógicas). Estos últimos dos nodos se han separado del nodo aritmético para respetar las comprobaciones semánticas (creando una separación clara entre nodos *numéricos* y nodos *lógicos*).

El nodo **vacío** merece una mención especial. Como se indica en la tabla, el nodo vacío no es un nodo real del AST, sino una representación de algún error durante la creación del AST. En nuestro caso, el tratamiento que hemos hecho del nodo es el siguiente: si en cualquier momento durante la creación de un nodo se detecta cualquier tipo de error (ya sea sintáctico o semántico), el nodo que se fuese a crear será sustituido por un nodo vacío para simbolizar el error en el AST.

Para la **implementación** de los nodos del AST se han creado varias clases (una por nodo descrito previamente), con un funcionamiento muy sencillo. Para crear un nodo basta con llamar al constructor del nodo correspondiente pasándole como argumentos todos los atributos del nodo (excepto el tipo, que se calcula internamente). El constructor asignará todos los valores y llamará automáticamente al método **compsem** para realizar las comprobaciones semánticas (como se describió previamente)

Además, cada nodo incluye los métodos **arbol** y **calculaProfundidad(profundidad)**. El método **arbol** simplemente imprime el nodo y sus hijos en formato de *string*, y el método **calculaProfundidad** se usa para calcular la profundidad que tiene cada nodo en el AST, de cara a aplicar tabulaciones dentro del método **arbol**. En conjunto, ambos métodos se usan para imprimir el AST resultante tras su construcción.

Toda la implementación se encuentra detallada en el fichero *AST.py*.

4.3.2. Construcción del AST e implementación

La **construcción** del AST se ha realizado insertando instrucciones directamente en la gramática incontextual, construyendo en cada producción el nodo apropiado del árbol. El árbol se construye de forma **bottom-up** (empezando a construirlo por los nodos hoja), y se construirá a partir del no terminal *instrucciones* (ya que la declaración de variables no forma parte del AST)

Concretamente, las instrucciones incluidas en la gramática para la construcción del árbol (sin contar las de las restricciones semánticas) son las siguientes:

1. **<Programa>** → PROGRAMA id ; <decl_var> <instrucciones> .
 {si ERROR entonces <Programa>.arb = NodoVacio(token.linea)
 Sino <Programa>.arb = <instrucciones>.arb }
2. **<instrucciones>** → INICIO <lista_inst> FIN
 {si ERROR entonces <instrucciones>.arb = NodoVacio(token.linea)
 sino <instrucciones>.arb = NodoCompuesta(<lista_inst>.lista, token.linea) }
3. **<lista_inst>** → <instrucción> ; <lista_inst>₁
 {si ERROR entonces <lista_inst>.lista = [NodoVacio(token.linea)] + <lista_inst>₁.lista
 sino <lista_inst>.lista = [<instrucción>.arb] ++ <lista_inst>₁.lista }
4. **<lista_inst>** → λ
 {si ERROR entonces <lista_inst>.lista = [NodoVacio(token.linea)]
 sino <lista_inst>.lista = []}
5. **<instrucción>** → INICIO <lista_inst> FIN
 {si ERROR entonces <instrucción>.arb = NodoVacio(token.linea)
 sino <instrucción>.arb = NodoCompuesta(<lista_inst>.lista, token.linea)}
6. **<instrucción>** → <inst_simple>
 {si ERROR entonces <instrucción>.arb = NodoVacio(token.linea)
 sino <instrucción>.arb = <inst_simple>.arb }
7. **<instrucción>** → <inst_es>
 {si ERROR entonces <instrucción>.arb = NodoVacio(token.linea)
 sino <instrucción>.arb = <inst_es>.arb }
8. **<instrucción>** → SI <expresion> ENTONCES <instrucción>₁ SINO <instrucción>₂
 {si ERROR entonces <instrucción>.arb = NodoVacio(token.linea)
 sino <instrucción>.arb = NodoSi(<expresion>.arb, <instrucción>₁.arb, <instrucción>₂.arb, token.linea)}
9. **<instrucción>** → MIENTRAS <expresión> HACER <instrucción>₁
 {si ERROR entonces <instrucción>.arb = NodoVacio(token.linea)
 sino <instrucción>.arb = NodoMientras(<expresion>.arb, <instrucción>₁.arb, token.linea)}
10. **<inst_simple>** → id {<resto_instsimple>.h = id} <resto_instsimple>
 {si ERROR entonces <instr_simple>.arb = NodoVacio(token.linea)
 sino <instr_simple>.arb = <resto_instsimple>.arb }
11. **<resto_instsimple>** → opasigna <expresión>
 {si ERROR entonces <resto_instsimple>.arb = NodoVacio(token.linea)
 sino
 variable = NodoAccesoVariable(<resto_instsimple>.h.lexema, token.linea)
 <resto_instsimple>.arb = NodoAsignación(variable, <expresión>.arb, token.linea) }

12. **<resto_instsimple>** → [<expr_simple>] opasigna <expresión>
 {si ERROR entonces <resto_instsimple>.arb = NodoVacio(token.linea)
 sino
 vector = NodoAccesoVector(<resto_instsimple>.h.lexema, <expr_simple>.arb,
 token.linea, <resto_instsimple>.h.tipo)
 <resto_instsimple>.arb = NodoAsignación(vector, <expresión>.arb, token.linea) }
13. **<resto_instsimple>** → λ
 {si ERROR entonces <resto_instsimple>.arb = NodoVacio(token.linea)
 sino
 <resto_instsimple>.arb = NodoAccesoVariable(<resto_instsimple>.h.lexema, token.linea) }
14. **<variable>** → id {<resto_var>.h = id} <resto_var>
 {si ERROR entonces <variable>.arb = NodoVacio(token.linea)
 sino <variable>.arb = <resto_var>.arb }
15. **<resto_var>** → [<expr_simple>]
 {si ERROR entonces <resto_var>.arb = NodoVacio(token.linea)
 sino <resto_var>.arb = NodoAccesoVector(<resto_var>.h.lexema, <expresión>.arb,
 token.linea)}
16. **<resto_var>** → λ
 {si ERROR entonces <resto_var>.arb = NodoVacio(token.linea)
 sino <resto_var>.arb = NodoAccesoVariable(<resto_var>.h.lexema, token.linea)}
17. **<inst_es>** → LEE (id)
 {si ERROR entonces <inst_es>.arb = NodoVacio(token.linea)
 sino
 var = NodoAccesoVariable(id.lexema, token.linea)
 <inst_es>.arb = NodoLee(var, token.linea)}
18. **<inst_es>** → ESCRIBE (<expr_simple>)
 {si ERROR entonces <inst_es>.arb = NodoVacio(token.linea)
 sino <inst_es>.arb = NodoEscribe(<expr_simple>.arb, token.linea)}
19. **<expresión>** → <expr_simple> {<expresiónPrime>.h = <expr_simple>.arb}
 <expresiónPrime>
 {si ERROR entonces <expresion>.arb = NodoVacio(token.linea)
 sino <expresion>.arb = <expresionPrime>.arb }
20. **<expresiónPrime>** → oprel <expr_simple>
 {si ERROR entonces <expresionPrime>.arb = NodoVacio(token.linea)
 sino <expresionPrime>.arb = NodoComparacion(<expresionPrime>.h, <expr_simple>.arb,
 oprel.operacion, token.linea) }
21. **<expresiónPrime>** → λ
 {si ERROR entonces <expresionPrime>.arb = NodoVacio(token.linea)
 sino <expresionPrime>.arb = <expresionPrime>.h }
22. **<expr_simple>** → <término> {<resto_exprsimple>.h = <término>.arb} <resto_exprsimple>
 {si ERROR entonces <expr_simple>.arb = NodoVacio(token.linea)
 sino <expr_simple>.arb = <resto_exprsimple>.arb }
23. **<expr_simple>** → <signo> <término> {<resto_exprsimple>.h = <término>.arb}
 <resto_exprsimple>
 {si ERROR entonces <expr_simple>.arb = NodoVacio(token.linea)
 sino <expr_simple>.arb = NodoSigno(<signo>.signo, <resto_exprsimple>.arb, token.linea) }
24. **<resto_exprsimple>** → opsuma <término> {<resto_exprsimple>₁.h =
 NodoAritmetico(<resto_exprsimple>.h, <término>.arb, opsuma.operacion, token.linea) }
 <resto_exprsimple>₁
 {si ERROR entonces <resto_exprsimple>.arb = NodoVacio(token.linea)
 sino <resto_exprsimple>.arb = <resto_exprsimple>₁.arb }

25. **<resto_exprsimple>** → O <término> { <resto_exprsimple>₁.h =
NodoLogico(<resto_exprsimple>.h, <término>.arb, "O", token.linea) } <resto_exprsimple>₁
{si ERROR entonces <resto_exprsimple>.arb = NodoVacio(token.linea)
sino <resto_exprsimple>.arb = <resto_exprsimple>₁.arb}
26. **<resto_exprsimple>** → λ
{si ERROR entonces <resto_exprsimple>.arb = NodoVacio(token.linea)
sino <resto_exprsimple>.arb = <resto_exprsimple>.h}
27. **<término>** → <factor> {<resto_term>.h = <factor>.arb} <resto_term>
{si ERROR entonces <término>.arb = NodoVacio(token.linea)
sino <término>.arb = <resto_term>.arb}
28. **<resto_term>** → opmult <factor> {<resto_term>₁.h = *NodoAritmetico(<resto_term>.h,*
<factor>.arb, opmult.operacion, token.linea) } <resto_term>₁
{si ERROR entonces <resto_exsimple>.arb = NodoVacio(token.linea)
sino <resto_term>.arb = <resto_term>₁.arb}
29. **<resto_term>** → Y <factor> {<resto_term>₁.h = *NodoLogico(<resto_term>.h, <factor>.arb,*
"Y", token.linea) } <resto_term>₁
{si ERROR entonces <resto_term>.arb = NodoVacio(token.linea)
sino <resto_term>.arb = <resto_term>₁.arb}
30. **<resto_term>** → λ
{si ERROR entonces <resto_term>.arb = NodoVacio(token.linea)
sino <resto_term>.arb = <resto_term>.h}
31. **<factor>** → <variable>
{si ERROR entonces <factor>.arb = NodoVacio(token.linea)
sino <factor>.arb = <variable>.arb }
32. **<factor>** → num
{si ERROR entonces <factor>.arb = NodoVacio(token.linea)
sino si num.tipo = "entero" entonces <factor>.arb = NodoEntero(num.valor, token.linea)
sino <factor>.arb = NodoReal(num.valor, token.linea)}
33. **<factor>** → (<expresión>)
{si ERROR entonces <factor>.arb = NodoVacio(token.linea)
sino <factor>.arb = <expresión>.arb }
34. **<factor>** → NO <factor>₁
{si ERROR entonces <factor>.arb = NodoVacio(token.linea)
sino <factor>.arb = NodoNo(<factor>₁.arb, token.linea) }
35. **<factor>** → CIERTO
{si ERROR entonces <factor>.arb = NodoVacio(token.linea)
sino <factor>.arb = NodoBooleano("cierto", token.linea) }
36. **<factor>** → FALSO
{si ERROR entonces <factor>.arb = NodoVacio(token.linea)
sino <factor>.arb = NodoBooleano("falso", token.linea) }
37. **<signo>** → + {<signo>.signo = "+"}
38. **<signo>** → - {<signo>.signo = "-"}

Tras todo el proceso de construcción, el árbol se encontrará almacenado en <Programa>.arbol, por lo que bastará con acceder a ese atributo para poder imprimir el árbol.

Para la **implementación** de la construcción del árbol, se ha realizado una modificación en los métodos para cada no terminal del analizador descendente recursivo.

Hemos añadido a cada método (excepto algunos casos concretos donde no era necesario) un booleano **error** (que comienza con valor True). Este booleano sirve para comprobar si ha habido un error sintáctico en algún punto del procesamiento de la producción, aunque posteriormente se haya “resuelto” sincronizando en modo pánico. También hemos creado un entero **línea**, que equivale a la línea del primer token de la producción, para usarlo en la construcción de los árboles.

Lo primero que se hace ahora en el método es crear un nodo vacío y asignarlo al atributo *arbol* del no terminal (representando los atributos con la clase **Atributos** descrita en el apartado de restricciones semánticas). De esta forma, si hubiese cualquier tipo de error (no se encuentran primeros, por ejemplo) nos aseguramos de que el nodo vacío ya está incluido. Durante la comprobación de la producción, si no encontramos algún terminal que se esperaba procedemos a cambiar el valor de error a False.

Al final del método del no terminal, si hemos conseguido analizar una producción entera con éxito y error sigue siendo True (no ha habido ningún error sintáctico), ya se procede a la construcción del nodo apropiado para la instrucción (o a la asignación del valor de algún no terminal de la producción como valor del árbol). Si este nodo creado tiene algún error semántico, volverá a ser sustituido por un nodo vacío.

Una vez se ha construido el árbol entero, este se imprimirá después de todos los errores que haya encontrado el analizador (llamando previamente al método **calculaProfundidad** del nodo de la raíz para poder tener un output claro)

4.4. Ejemplos

Para demostrar el correcto funcionamiento del analizador semántico, se incluirán una serie de ejemplos demostrando su funcionamiento.

El **primer ejemplo** se corresponde con una versión adaptada del programa que se nos ha suministrado como ejemplo. El programa (contenido en el fichero *pruebaProfesor.txt*) es el siguiente:

```
PROGRAMA p1; //COMENTARIO
VAR i,x:ENTERO;
INICIO
    i:=0;
    MIENTRAS (i<>5) HACER
        INICIO
            x:=i*4;
            ESCRIBE(x);
        FIN; //COMENTARIO
FIN.
```

La salida del analizador se puede observar en la siguiente figura. El programa se analiza sin encontrar ningún error y el AST obtenido es coherente y representa adecuadamente a la entrada.

```
Este es tu fichero 'pruebaProfesor.txt'
Análisis sintáctico/semántico SATISFACTORIO. Fichero : pruebaProfesor.txt CORRECTO

AST generado:
("Compuesta" "línea: 3"
  ("Asignación" "línea: 4"
    ("AccesoVariable" "var: i" "tipo: entero" "línea: 4")
    ("Entero" "valor: 0" "tipo: entero" "línea: 4")
  )
  ("Mientras" "línea: 5"
    ("Comparación" "op: <>" "tipo: booleano" "línea: 5"
      ("AccesoVariable" "var: i" "tipo: entero" "línea: 5")
      ("Entero" "valor: 5" "tipo: entero" "línea: 5")
    )
  )
  ("Compuesta" "línea: 6"
    ("Asignación" "línea: 7"
      ("AccesoVariable" "var: x" "tipo: entero" "línea: 7")
      ("Aritmética" "op: *" "tipo: entero" "línea: 7"
        ("AccesoVariable" "var: i" "tipo: entero" "línea: 7")
        ("Entero" "valor: 4" "tipo: entero" "línea: 7")
      )
    )
  )
  ("Escribe" "línea: 8"
    ("AccesoVariable" "var: x" "tipo: entero" "línea: 8")
  )
)
)
```

Ilustración 6. Análisis semántico del programa pruebaProfesor.

El **segundo ejemplo** se corresponde con un programa distinto, creado para intentar probar estructuras más complejas. El programa (contenido en el fichero *pruebaCorrecto.txt*) es el siguiente:

```
PROGRAMA p1;
VAR a : VECTOR [5] DE BOOLEANO;
    b, c, d: REAL;
INICIO
  a[1] := 1.1 < 2.2;
  MIENTRAS a[2] HACER b := (c*d) + 2;
  SI a[3] O a[4] ENTONCES
    c := -1.1
  SINO
    d := 0.0;
FIN.//COMENTARIO DE PRUEBA
```

La salida del analizador se puede observar en la siguiente figura. De nuevo, el programa se analiza sin encontrar ningún error y el AST se crea correctamente y de forma coherente.

```

Este es tu fichero 'pruebaCorrecto.txt'
Análisis sintáctico/semántico SATISFACTORIO. Fichero : pruebaCorrecto.txt CORRECTO

AST generado:
("Compuesta" "línea: 4"
  ("Asignacion" "línea: 5"
    ("AccesoVector" "var: a" "tipo: booleano" "línea: 5"
      ("Entero" "valor: 1" "tipo: entero" "línea: 5")
    )
    ("Comparacion" "op: <" "tipo: booleano" "línea: 5"
      ("Real" "valor: 1.1" "tipo: real" "línea: 5")
      ("Real" "valor: 2.2" "tipo: real" "línea: 5")
    )
  )
  ("Mientras" "línea: 6"
    ("AccesoVector" "var: a" "tipo: booleano" "línea: 6"
      ("Entero" "valor: 2" "tipo: entero" "línea: 6")
    )
    ("Asignacion" "línea: 6"
      ("AccesoVariable" "var: b" "tipo: real" "línea: 6")
      ("Aritmetica" "op: +" "tipo: real" "línea: 6"
        ("Aritmetica" "op: *" "tipo: real" "línea: 6"
          ("AccesoVariable" "var: c" "tipo: real" "línea: 6")
          ("AccesoVariable" "var: d" "tipo: real" "línea: 6")
        )
      )
      ("Entero" "valor: 2" "tipo: entero" "línea: 6")
    )
  )
  ("Si" "línea: 7"
    ("Logica" "op: o" "tipo: booleano" "línea: 7"
      ("AccesoVector" "var: a" "tipo: booleano" "línea: 7"
        ("Entero" "valor: 3" "tipo: entero" "línea: 7")
      )
      ("AccesoVector" "var: a" "tipo: booleano" "línea: 7"
        ("Entero" "valor: 4" "tipo: entero" "línea: 7")
      )
    )
    ("Asignacion" "línea: 8"
      ("AccesoVariable" "var: c" "tipo: real" "línea: 8")
      ("Signo" "signo: -" "tipo: real" "línea: 8"
        ("Real" "valor: 1.1" "tipo: real" "línea: 8")
      )
    )
    ("Asignacion" "línea: 10"
      ("AccesoVariable" "var: d" "tipo: real" "línea: 10")
      ("Real" "valor: 0.0" "tipo: real" "línea: 10")
    )
  )
)

```

Ilustración 7. Análisis semántico del programa *pruebaCorrecto*.

El **último** ejemplo se corresponde con un programa que contiene algunos errores semánticos, buscando comprobar el funcionamiento en caso de errores semánticos. El programa (contenido en el fichero *pruebaErrorSemantico.txt*) es el siguiente:

```

PROGRAMA p1; //COMENTARIO DE PRUEBA
VAR a: BOOLEANO;
    b, a, programa: ENTERO;

```



```
INICIO
  MIENTRAS 1 + 2 HACER
    INICIO
      b := 1 < 2;
      a[1] := CIERTO;
    FIN;
  FIN.
```

Podemos observar que el programa contiene numerosos errores semánticos (incluyendo variables duplicadas y tipos incompatibles). La salida del analizador se puede encontrar en la figura siguiente. Podemos observar que ha detectado todos los errores correctamente, y que ha incluido nodos vacíos en vez de los nodos correspondientes.

```
Este es tu fichero 'pruebaErrorSemantico.txt'
Linea: 3 ERROR: El identificador a ya ha sido declarado previamente
Linea: 3 ERROR: El identificador programa esta tomando un valor prohibido (los identificadores no pueden tomar valor
es de palabras reservadas)
Linea: 7 ERROR: El valor que se intenta asignar a la variable no es compatible con el tipo de la variable
Linea: 8 ERROR: Se esperaba un vector, el identificador a no lo es
Linea: 5 ERROR: Se esperaba una condicion de tipo logico
Análisis sintactico/semantico CON ERRORES. Fichero : pruebaErrorSemantico.txt ERRONEO

AST generado (con posibles errores):
("Compuesta" "linea: 4"
  ("Vacio (ERROR)" "linea: 5")
)
```

Ilustración 8. Análisis semántico del programa pruebaErrorSemantico.

5. Conclusiones

Ha sido una práctica en la que hemos aprendido como programar un compilador y los problemas que surgen cuando intentas hacerlo.

El desarrollo de la practica nos ha ayudado a reforzar bastante los conceptos vistos en clase y a interiorizarlos. Como prácticamente todas las cosas, la teoría es más sencilla que la práctica, pues a la hora de aplicarla (construir el compilador en sí) es bastante tedioso hacer todos los pasos explicados en clase funcionen correctamente, y surgen numerosos problemas inesperados.

También surgen problemas que en la teoría no se dan, pues hay que lidiar con las limitaciones del lenguaje de programación sobre el que se está trabajando y a veces se tiene que arreglar cosas de forma poco convencional.

Dado todo esto creemos que, en esta asignatura, hacer este tipo de práctica es obligatorio para poder entender bien cómo funciona realmente un compilador, y saber crear uno.

A nuestro grupo le ha llevado bastante más tiempo del que esperábamos hacer la práctica, la mayor parte de éste dedicado al analizador sintáctico y semántico. Aun así, ha sido entretenido, interesante y didáctico el tiempo que hemos dedicado a realizarlo.

Si hay algún aspecto a mejorar de cara a cursos posteriores es el guion de la práctica. Resulta bastante útil para realizar el analizador léxico, pero en la parte del analizador sintáctico (la parte de errores y modo pánico, sobre todo) y semántico hay algunas cosas que no quedan demasiado claras y algunas otras que se desarrollan menos de lo que deberían.