Procesadores de Lenguaje – 2019/2020

**Creación de un compilador**

Práctica 3

Jesús Andrés Fernández

Luna Jiménez Fernández

Índice

[1. Introducción 4](#_Toc29436560)

[2. Analizador léxico 4](#_Toc29436561)

[2.1. Especificación formal del lenguaje 4](#_Toc29436562)

[2.2. Máquina Discriminadora Determinista (MDD) asociada 4](#_Toc29436563)

[2.3. Implementación 4](#_Toc29436564)

[3. Analizador sintáctico 4](#_Toc29436565)

[3.1. Producciones del lenguaje 4](#_Toc29436566)

[3.2. Comprobación y transformación de la gramática a LL(1) 4](#_Toc29436567)

[3.3. Primeros y siguientes de la gramática 4](#_Toc29436568)

[3.4. Implementación 4](#_Toc29436569)

[4. Analizador semántico 4](#_Toc29436570)

[4.1. Tabla de símbolos e implementación 4](#_Toc29436571)

[4.2. Restricciones semánticas e implementación 5](#_Toc29436572)

[4.2.1. Restricciones semánticas sobre la gramática 6](#_Toc29436573)

[4.2.2. Restricciones semánticas sobre el AST 7](#_Toc29436574)

[4.3. Construcción del árbol de Sintaxis Abstracta (AST) e implementación 8](#_Toc29436575)

[4.3.1. Nodos del AST e implementación 8](#_Toc29436576)

[4.3.2. Construcción del AST e implementación 10](#_Toc29436577)

[4.4. Ejemplos 13](#_Toc29436578)

[5. Conclusiones 16](#_Toc29436579)

Índice de ilustraciones

[Ilustración 1. Análisis semántico del fichero pruebaProfesor. 14](file:///D:\GitHub\PL_Pract3-G9-19_20\Memoria\Practica%20-%20Creacion%20de%20un%20compilador%20-%20Jesus%20y%20Luna.docx#_Toc29436525)

[Ilustración 2. Análisis semántico del fichero pruebaCorrecto. 15](file:///D:\GitHub\PL_Pract3-G9-19_20\Memoria\Practica%20-%20Creacion%20de%20un%20compilador%20-%20Jesus%20y%20Luna.docx#_Toc29436526)

[Ilustración 3. Salida del analizador semántico para pruebaProfesor. 15](file:///D:\GitHub\PL_Pract3-G9-19_20\Memoria\Practica%20-%20Creacion%20de%20un%20compilador%20-%20Jesus%20y%20Luna.docx#_Toc29436527)

[Ilustración 4. Análisis semántico del programa pruebaErrorSemantico. 16](file:///D:\GitHub\PL_Pract3-G9-19_20\Memoria\Practica%20-%20Creacion%20de%20un%20compilador%20-%20Jesus%20y%20Luna.docx#_Toc29436528)

Índice de tablas

[Tabla 1. Nodos del AST 9](#_Toc29436530)

# 1. Introducción

# 2. Analizador léxico

## 2.1. Especificación formal del lenguaje

## 2.2. Máquina Discriminadora Determinista (MDD) asociada

## 2.3. Implementación

# 3. Analizador sintáctico

## 3.1. Producciones del lenguaje

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

## 3.3. Primeros y siguientes de la gramática

## 3.4. Implementación

# 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> .

1. **<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>

1. **<decl\_var>** → λ
2. **<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>

1. **<decl\_v>** → λ
2. **<lista\_id>** → id *{<resto\_listaid>.lh = [id.lexema]}* <resto\_listaid>

*{<lista\_id>.lista = <resto\_listaid>.lista}*

1. **<resto\_listaid>** → , <lista\_id> *{<resto\_listaid>.lista = <resto\_listaid>.lh ++ <lista\_id>.lista}*
2. **<resto\_listaid>** → λ *{<resto\_listaid>.lista = <resto\_listaid>.lh}*
3. **<Tipo>** → <tipo\_std> *{<Tipo>.t = <tipo\_std>.t; <Tipo>.clase = “variable”, <Tipo>.longitud = 0}*
4. **<Tipo>** → VECTOR [num] DE <Tipo\_std> *{<Tipo>.t = <tipo\_std>.t; <Tipo>.clase = “vector”; <Tipo>.longitud = num.valor}*
5. **<Tipo\_std>** → ENTERO *{<tipo\_std>.t = “entero”}*
6. **<Tipo\_std>** → REAL *{<tipo\_std>.t = “real”}*
7. **<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 | --- | * Símbolo al que se asigna * Expresión a asignar |
| Si | Sentencia condicional SI \_ ENTONCES \_ SINO \_ | --- | * Condición * Instrucciones para el SI * Instrucciones para el SINO |
| Mientras | Sentencia de bucle MIENTRAS \_ HACER \_ | --- | * Condición * Instrucciones |
| Lee | Sentencia de lectura de una variable simple | * Variable simple a leer | --- |
| Escribe | Sentencia de escritura en pantalla | --- | * Expresión a escribir |
| Compuesta | Sentencia compuesta; lista de instrucciones | --- | * Lista de instrucciones |
| Comparación | Comparación de expresiones | * Tipo resultante * Operación realizada | * Dos expresiones a comparar |
| Aritmético | Operación aritmética entre expresiones | * Tipo resultante * Operación realizada | * Dos expresiones a operar |
| Entero | Valor de tipo entero | * Tipo resultante * Valor | --- |
| Real | Valor de tipo real | * Tipo resultante * Valor | --- |
| Booleano | Valor de tipo booleano | * Tipo resultante * Valor | --- |
| AccesoVariable | Acceso a una variable simple | * Tipo resultante * Identificador de la variable | --- |
| AccesoVector | Acceso a un vector | * Tipo resultante * Identificador del vector | --- |
| Signo | Signo aplicado a una expresión | * Tipo resultante * Signo utilizado | * Expresión sobre la que aplicar el signo |
| Lógico | Operación lógica entre expresiones | * Tipo resultante * Operación realizada | * Dos expresiones a operar |
| No | Operación lógica NO sobre una expresión | * Tipo resultante | * Expresión sobre la que aplicar el NO lógico |
| Vacío | Nodo especial que simboliza error sintáctico o semántico durante la construcción del árbol | --- | --- |

Tabla . 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 }*

1. **<instrucciones>** → INICIO <lista\_inst> FIN

*{si ERROR entonces <instrucciones>.arb = NodoVacio(token.linea)*

*sino <instrucciones>.arb = NodoCompuesta(<lista\_inst>.lista, token.linea) }*

1. **<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 }*

1. **<lista\_inst>** → λ

*{si ERROR entonces <lista\_inst>.lista = [NodoVacio(token.linea)]*

*sino <lista\_inst>.lista = []}*

1. **<instrucción>** → INICIO <lista\_inst> FIN

*{si ERROR entonces <instrucción>.arb = NodoVacio(token.linea)*

*sino <instrucción>.arb = NodoCompuesta(<lista\_inst>.lista, token.linea)}*

1. **<instrucción>** → <inst\_simple>

*{si ERROR entonces <instrucción>.arb = NodoVacio(token.linea)*

*sino <instrucción>.arb = <inst\_simple>.arb}*

1. **<instrucción>** → <inst\_es>

*{si ERROR entonces <instrucción>.arb = NodoVacio(token.linea)*

*sino<instrucción>.arb = <inst\_es>.arb}*

1. **<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)}*

1. **<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)}*

1. **<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 }*

1. **<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) }*

1. **<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) }*

1. **<resto\_instsimple>** → λ

*{si ERROR entonces <resto\_instsimple>.arb = NodoVacio(token.linea)*

*sino*

*<resto\_instsimple>.arb = NodoAccesoVariable(<resto\_instsimple>.h.lexema, token.linea) }*

1. **<variable>** → id *{<resto\_var>.h = id}* <resto\_var>

*{si ERROR entonces <variable>.arb = NodoVacio(token.linea)*

*sino <variable>.arb = <resto\_var>.arb }*

1. **<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)}*

1. **<resto\_var>** → λ

*{si ERROR entonces <resto\_var>.arb = NodoVacio(token.linea)*

*sino <resto\_var>.arb = NodoAccesoVariable(<resto\_var>.h.lexema, token.linea)}*

1. **<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)}*

1. **<inst\_es>** → ESCRIBE ( <expr\_simple>)

*{si ERROR entonces <inst\_es>.arb = NodoVacio(token.linea)*

*sino <inst\_es>.arb = NodoEscribe(<expr\_simple>.arb, token.linea)}*

1. **<expresión>** → <expr\_simple> {<expresiónPrime>.h = <expr\_simple>.arb} <expresiónPrime>

*{si ERROR entonces <expresion>.arb = NodoVacio(token.linea)*

*sino <expresion>.arb = <expresionPrime>.arb}*

1. **<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) }*

1. **<expresiónPrime>** → λ

*{si ERROR entonces <expresionPrime>.arb = NodoVacio(token.linea)*

*sino <expresionPrime>.arb = <expresionPrime>.h}*

1. **<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}*

1. **<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) }*

1. **<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}*

1. **<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}*

1. **<resto\_exprsimple>** → λ

*{si ERROR entonces <resto\_exprsimple>.arb = NodoVacio(token.linea)*

*sino <resto\_exprsimple>.arb = <resto\_exprsimple>.h}*

1. **<término>** → <factor> *{<resto\_term>.h = <factor>.arb}* <resto\_term>

*{si ERROR entonces <término>.arb = NodoVacio(token.linea)*

*sino <término>.arb = <resto\_term>.arb}*

1. **<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}*

1. **<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}*

1. **<resto\_term>** → λ

*{si ERROR entonces <resto\_term>.arb = NodoVacio(token.linea)*

*sino <resto\_term>.arb = <resto\_term>.h}*

1. **<factor>** → <variable>

*{si ERROR entonces <factor>.arb = NodoVacio(token.linea)*

*sino <factor>.arb = <variable>.arb }*

1. **<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)}*

1. **<factor>** → ( <expresión> )

*{si ERROR entonces <factor>.arb = NodoVacio(token.linea)*

*sino <factor>.arb = <expresión>.arb }*

1. **<factor>** → NO <factor>1

*{si ERROR entonces <factor>.arb = NodoVacio(token.linea)*

*sino <factor>.arb = NodoNo(<factor>1.arb, token.linea) }*

1. **<factor>** → CIERTO

*{si ERROR entonces <factor>.arb = NodoVacio(token.linea)*

*sino <factor>.arb = NodoBooleano(“cierto”, token.linea) }*

1. **<factor>** → FALSO

*{si ERROR entonces <factor>.arb = NodoVacio(token.linea)*

*sino <factor>.arb = NodoBooleano(“falso”, token.linea) }*

1. **<signo>** → + *{<signo>.signo = “+”}*
2. **<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.

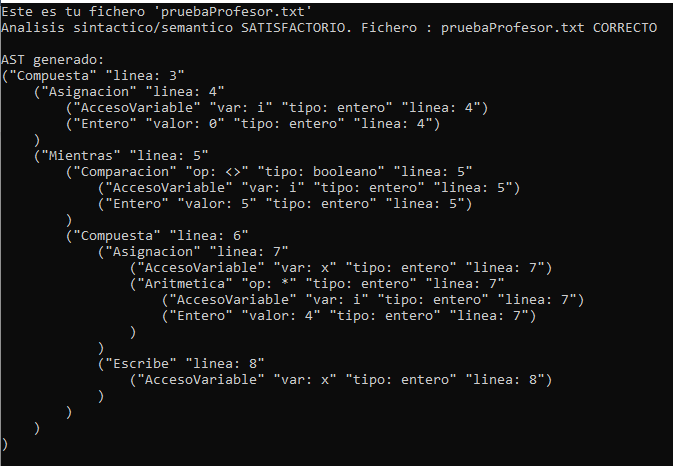
La salida del analizador se puede observar en la siguiente figura. El programa se analiza sin encontrar ningún error y el AST obtenido es coherente y representa adecuadamente a la entrada.

Ilustración . Análisis semántico del fichero 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

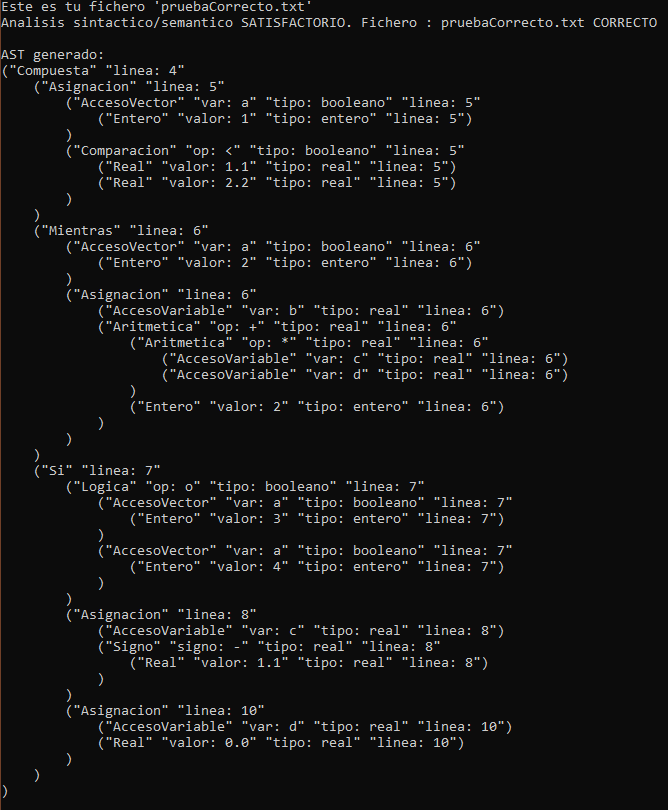
La salida del analizador se puede observar en la siguiente figura. De nuevo, el programa se analiza sin encontrar ningún error y el AST se crea correctamente y de forma coherente.

Ilustración . Análisis semántico del fichero pruebaCorrecto.

Ilustración 3. Salida del analizador semántico para pruebaProfesor.

El ultimo 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.

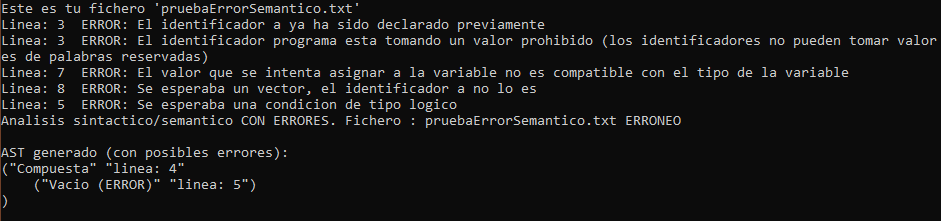
Podemos observar que el programa contiene numerosos errores semánticos (incluyendo variables duplicadas y tipos incompatibles). La salida del analizador se puede encontrar en la figura siguiente. Podemos observar que ha detectado todos los errores correctamente, y que ha incluido nodos vacíos en vez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

Ilustración . Análisis semántico del programa pruebaErrorSemantico.

# 5. Conclusiones