Procesadores de Lenguaje – 2019/2020

**Creación de un compilador**

Práctica 3

Jesús Andrés Fernández

Luna Jiménez Fernández

Índice

[1. Introducción 3](#_Toc29349215)

[2. Analizador léxico 3](#_Toc29349216)

[2.1. Especificación formal del lenguaje 3](#_Toc29349217)

[2.2. Máquina Discriminadora Determinista (MDD) asociada 3](#_Toc29349218)

[2.3. Implementación 3](#_Toc29349219)

[3. Analizador sintáctico 3](#_Toc29349220)

[3.1. Producciones del lenguaje 3](#_Toc29349221)

[3.2. Comprobación y transformación de la gramática a LL(1) 3](#_Toc29349222)

[3.3. Primeros y siguientes de la gramática 3](#_Toc29349223)

[3.4. Implementación 3](#_Toc29349224)

[4. Analizador semántico 3](#_Toc29349225)

[4.1. Tabla de símbolos e implementación 3](#_Toc29349226)

[4.2. Restricciones semánticas e implementación 4](#_Toc29349227)

[4.2.1. Restricciones semánticas sobre la gramática 5](#_Toc29349228)

[4.2.2. Restricciones semánticas sobre el AST 6](#_Toc29349229)

[4.3. Construcción del árbol de Sintaxis Abstracta (AST) e implementación 7](#_Toc29349230)

[5. Conclusiones 7](#_Toc29349231)

# 1. Introducción

# 2. Analizador léxico

En este apartado se discutirá el desarrollo del analizador sintáctico utilizado en el compilador a implementar. Concretamente se discutirán especificación formal del lenguaje, la Máquina Discriminadora Determinista Asociada y una descripción de como se ha realizado la implementación.

## 2.1. Especificación formal del lenguaje

A continuación, vamos a ver la especificación del lenguaje en una tabla formada por las categorías que tiene, la expresión regular de cada una, qué hacer con ellas 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.

|  |  |  |  |
| --- | --- | --- | --- |
| 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 | [ \t]+ | omitir |  |
| Salto de linea | [/n]+ | omitir |  |
| EOF | EOF | emitir |  |
| Operador Asignación | := | emitir |  |
| Operador Suma | [+-] | copiar operación | operación |
| Operador Multiplicación | [\*/] | copiar operación | operación |
| Numero | [0-9][0-9]\*(.[0-9][0-9]\*)? | copiar tipo y valor | valor  tipo |
| Identificador | [a-zA-Z][a-zA-Z0-9]\* | copiar valor | valor |
| Operador Relacional | **=**|<>|<|<=|>=|> | copiar operación | operación |
| Paréntesis de Apertura | ( | emitir |  |
| Paréntesis de Cierre | ) | emitir |  |
| Corchete de Apertura | [ | emitir |  |
| Corchete de Cierre | ] | emitir |  |
| Punto | . | emitir |  |
| Dos Puntos | : | emitir |  |
| Punto y Coma | ; | emitir |  |
| Coma | , | emitir |  |

## 2.2. Máquina Discriminadora Determinista (MDD) asociada

## 2.3. Implementación

Ahora vamos a ver como se ha realiza la implementación del analizador léxico. Para iniciarla se le pasa un fichero sobre el que se abre un flujo. Tras esto se llama al método ***Analiza()*** sobre el flujo anteriormente mencionado, lo que comienza el proceso. Éste método nos devuelve el componente léxico que ha leído en la entrada, y sigue buscando componentes hasta que se encuentra el finde fichero *EOF*.

Dentro de este método lo que hace es implementar la MDD descrita en el apartado anterior, mediante flujo de control(estructuras *if/else*) va comprobando los caracteres y pasando de un sitio a otro conforme corresponda. Por ejemplo, al encontrar un “.” emite un elemento de la categoría léxica punto.

Debemos además especificar cómo esperamos que se unan las categorías para formar la entrada. Se va a implementar la “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 además métodos aparte para tratar 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*).
* **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 nombre del identificador/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) tiene 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 incluídas en un *FrozenSet*). Si lo está emitimos la palabra reservada correspondiente, y si no lo está, un identificador.

# 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.

## 3.1. Producciones del lenguaje

A continuación, se mostrarán las producciones 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 <expresion> 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 recursión 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 ya no hay prefijos comunes ni recursividad a izquierdas, podemos afirmar que la gramática es LL(1).

## 3.3. Primeros y siguientes de la gramática

### 3.3.1. Primeros

A continuación, vamos a obtener el conjunto de primeros de cada regla y señalar si son anulables o no:

1. **<Programa>** → PROGRAMA id ; <decl\_var> <instrucciones> .

*{PROGRAMA}*

1. **<decl\_var>** → VAR <lista\_id> : <tipo> ; <decl\_v>

*{VAR}*

1. **<decl\_var>** → λ

*{}* ***ANULABLE***

1. **<decl\_v>** → <lista\_id> : <tipo> ; <decl\_v>

*{id}*

1. **<decl\_v>** → λ

*{}* ***ANULABLE***

1. **<lista\_id>** → id <resto\_listaid>

*{id}*

1. **<resto\_listaid>** → , <lista\_id>

*{,}*

1. **<resto\_listaid>** → λ

*{}* ***ANULABLE***

1. **<Tipo>** → <tipo\_std>

*{ENTERO, REAL, BOOLEANO}*

1. **<Tipo>** → VECTOR [num] DE <Tipo\_std>

*{VECTOR}*

1. **<Tipo\_std>** → ENTERO

*{ENTERO}*

1. **<Tipo\_std>** → REAL

*{REAL}*

1. **<Tipo\_std>** → BOOLEANO

*{BOOLEANO}*

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

*{INICIO}*

1. **<lista\_inst>** → <instrucción> ; <lista\_inst>

*{id, INICIO, LEE, ESCRIBE, SI, MIENTRAS}*

1. **<lista\_inst>** → λ

*{}* ***ANULABLE***

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

*{INICIO}*

1. **<instrucción>** → <inst\_simple>

*{id}*

1. **<instrucción>** → <inst\_es>

*{LEE, ESCRIBE}*

1. **<instrucción>** →  SI <expresion> ENTONCES <instrucción> SINO <instrucción>

*{SI}*

1. **<instrucción>** →  MIENTRAS <expresión> HACER <instrucción>

*{MIENTRAS}*

1. **<inst\_simple>** → id <resto\_instsimple>

*{id}*

1. **<resto\_instsimple>** → opasigna <expresión>

*{opasigna}*

1. **<resto\_instsimple>** → [ <expr\_simple> ] opasigna <expresión>

*{ [ }*

1. **<resto\_instsimple>** → λ

*{}* ***ANULABLE***

1. **<variable>** → id <resto\_var>

*{id}*

1. **<resto\_var>** → [ <expr\_simple> ]

*{ [ }*

1. **<resto\_var>** → λ

*{}* ***ANULABLE***

1. **<inst\_es>** → LEE ( id )

*{LEE}*

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

*{ESCRIBE}*

1. **<expresión>** → <expr\_simple> <expresiónPrime>

*{id, num, (, NO, CIERTO, FALSO, +, -}*

1. **<expresiónPrime>** → oprel <expr\_simple>

*{oprel}*

1. **<expresiónPrime>** → λ

*{}* ***ANULABLE***

1. **<expr\_simple>** → <término> <resto\_exsimple>

*{id, num, (, NO, CIERTO, FALSO}*

1. **<expr\_simple>** → <signo> <término> <resto\_exsimple>

*{+, -}*

1. **<resto\_exsimple>** → opsuma <término> <resto\_exsimple>

*{opsuma}*

1. **<resto\_exsimple>** → O <término> <resto\_exsimple>

*{O}*

1. **<resto\_exsimple>** → λ

*{}* ***ANULABLE***

1. **<término>** → <factor> <resto\_term>

*{id, num, (, NO, CIERTO, FALSO}*

1. **<resto\_term>** → opmult <factor> <resto\_term>

*{opmult}*

1. **<resto\_term>** → Y <factor> <resto\_term>

*{Y}*

1. **<resto\_term>** → λ

*{}* ***ANULABLE***

1. **<factor>** → <variable>

*{id}*

1. **<factor>** → num

*{num}*

1. **<factor>** → ( <expresión> )

*{ ( }*

1. **<factor>** → NO <factor>

*{NO}*

1. **<factor>** → CIERTO

*{CIERTO}*

1. **<factor>** → FALSO

*{FALSO}*

1. **<signo>** → +

*{+}*

1. **<signo>** → -

*{-}*

### 3.3.2. Siguientes

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

* **Primer paso**: añadimos el fin de fichero al no terminal inicial:

|  |  |
| --- | --- |
| **<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 <A> → αρβ:

primeros(β) ⊆ siguientes(ρ)

|  |  |
| --- | --- |
| **<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 hacer:

Si <A> → αρβ y β es anulable:

siguientes(<A>) ⊆ siguientes(ρ)

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.

## 3.4. Implementación

Antes de nada, algunos métodos que se han usado para la implementación:

* **avanza()**: lee el siguiente componente léxico.
* **<NombreDeNoTerminal>()**:Analiza el no terminal que coincide con el nombre del método (Los parámetros no se nombran, pues son para la parte del analizador semántico).
* **Error(num, linea)**: imprime un mensaje con el error correspoindiente y pone a falso la variable ***“aceptacion”*** que hace que imprima que el análisis es insatisfactorio, pues tiene errores. El valor de ***“num”***determina cual es el error que debe imprimirse y ***“linea”*** representa la línea donde se ha producido ese error.
* **Sincroniza(categoriasSiguientes, reservadasSiguientes, categoria, reservada)**: Los parámetros “categoría” hacen referencia a las categorías léxicas, mientras que los “reservadas” son para distinguir elementos de la categoría “PalabraReservada”. Los parámetros “Siguientes” hacen referencia a los siguientes del no teminal que los llama, mientras que las que no tienen esta notación hacen referencia al elemento que los llama. Este método avanza componentes léxicos hasta encontrar uno de los que le hemos pasado en los parámetros. Si se trata de un elemento “Siguiente”, devuelve el elemento, mientras que si se trata de los otros, avanza una posición más antes de volver.

Para la implementación hemos seguido la siguiente estrategia:

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

Ejemplo: Al principio de llama a “***Programa()***” para analizar el no terminal inicial.

1. 2º - Se revisan los primeros de las reglas del no terminal. Si el componente actual coincide con alguno de los primeros de alguna regla, se elige 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 y si lo hace volvemos sin hacer nada. 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>)*.**

1. 3º - 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 es dicho elemento al comprobar los primeros de la regla, en caso contrario se llama al método de análisis del no terminal correspondiente. A partir de esto, cuando se espera un no terminal, se llama al método correspondiente, esperando a que termine para proseguir con el análisis. Si el siguiente elemento se trata de un terminal como siguiente elemento, se comprueba que es ese no terminal y se llama a ***avanza()***, siguiendo con el análisis. Si no se puede encontrar este no terminal, se lanza un mensaje de error con ***Error(num, linea)***.

Ejemplo: Una vez que hemos entrado en ***<inst\_es> → ESCRIBE ( <expr\_simple>)*** se llama a ***avanza()*** ya que hemos encontrado “ESCRIBE” 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 eventos, 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 dar 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 los primeros o siguientes (en caso de ser anulable), **sincronizamos** hasta encontrar un primero para poder ejecutar la regla o encontrar un siguiente y omitir el análisis de ese no terminal.

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 encontramos “LEE”, pasaremos a analizar la regla ***<inst\_es> → ESCRIBE ( <expr\_simple>)***, pero si encontramos “;”, haremos un return.

* Si el error nos lo encontramos en el 3º paso, esto es, que no puede encontrar el terminal que corresponde leer, **sincronizamos** hasta encontrar el terminal que corresponda a los siguientes del no terminal o el elemento que se esperaba encontrar. Si se obtiene el elemento que se esperaba, se continúa el análisis a partir de ahí, en caso contrario (siguientes del no terminal), vuelve 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 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, continuamos con el análisis desde ***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” dará un error especial de final de fichero inesperado y se detendrá el análisis de errores.

# 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, 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

[ESCRIBE AQUÍ EL MÉTODO CALCULAPROFUNDIDAD]

[COMENTA AQUÍ QUE SI HAY ERROR SINTáCTICO O SEMÁNTICO SE CAMBIA EL NODO POR NODOVACIO]

[PON UNA TABLA CON LOS NODOS INCLUIDOS Y SUS ATRIBUTOS]

# 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 ayuda 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 llevarlo a cabo es bastante tedioso hacer todos los pasos explicados en clase y surgen problemas inesperados. Dado todo esto creemos que en esta asignatura, hacer este tipo de práctica es obligatorio para entender bien como funciona un compilador y saber hacer uno.

También surgen problemas que en la teoría no se dan, pues también 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 manera poco convencional.

A nuestro grupo le ha llevado bastante más tiempo del que pensábamos que nos iba a llevar hacer la práctica, la mayor parte de éste dedicado al analizador sintáctico y semántico.

Creemos que el principal aspecto a mejorar es el guion de la práctica, pues 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.