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

## 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, 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