***Procesadores de Lenguajes Memoria Práctica 4***

**Equipo MultiProRobo:**

* **Hernán Indíbil de la Cruz Calvo**
* **Alejandro Martín Simón Sánchez**
* **Alejandro Zornoza Martínez**

***ÍNDICE***

[1. Introducción 1](#_Toc503716885)

[2. Analizador Léxico 2](#_Toc503716886)

[2.1. Categorías léxicas 2](#_Toc503716887)

[2.2. Descripción del MDD 2](#_Toc503716888)

[2.3. Descripción de los errores 3](#_Toc503716889)

[2.4. Funciones y recursos utilizados 4](#_Toc503716890)

[3. Analizador Sintáctico 5](#_Toc503716891)

[3.1. Tabla de análisis, condición LL(1) y errores 5](#_Toc503716892)

[3.2. Funciones utilizadas 23](#_Toc503716893)

[4. Analizador Semántico 24](#_Toc503716894)

[4.1. Comprobación de identificadores 24](#_Toc503716895)

[4.2. Comprobación de tipos 26](#_Toc503716896)

[4.3. Árbol AST 30](#_Toc503716897)

[5. Conclusiones 32](#_Toc503716898)

# Introducción

En esta práctica se implementarán las partes de analizador léxico, analizador sintáctico y analizador semántico de un compilador.

Para describir informalmente como es el lenguaje se describe a continuación como es el programa:

El objetivo es procesar un lenguaje que comienza con la palabra PROGRAMA y un identificador, a continuación, constan las siguientes partes:

* Declaración de variables globales. Es opcional.
* Declaración de funciones. Es opcional.
* Sentencias compuestas que representa el cuerpo del programa principal: conjunto de instrucciones que van entre las palabras INICIO y FIN, terminadas en punto y coma.

Se espera que en caso de no haber fallos se indique que ha ido todo correctamente, y en caso de encontrar un fallo indicar donde se encuentra y qué se esperaba.

Así en esta práctica se desarrolla lo siguiente:

* Analizador léxico (obligatorio)
* Analizador sintáctico (obligatorio)
  + Gestión de errores sintácticos (obligatorio)
* Analizador semántico
  + Componentes identificadores (obligatorio)
  + Gestión de tipos (opcional)
* Generación árbol AST (opcional)

# Analizador Léxico

# Categorías léxicas

Para realizar el analizador léxico lo primero que se ha realizado ha sido identificar los diferentes tipos de componentes que pueden aparecer en el programa:

Palabras clave = {PROGRAMA, VAR, VECTOR, ENTERO, REAL, BOOLEANO, PROC, FUNCION, INICIO, FIN, SI, ENTONCES, SINO, MIENTRAS, HACER, LEE, ESCRIBE, Y, O, NO, CIERTO}

OpRel = {=, <>, <, <=, >=, >}

OpAdd = {+, -}

OpMult = {\*, /}

OpAsigna = {:=}

Así el lenguaje es la unión de: OpRel + OpRel + OpAdd + OpMult + OpAsigna + [a-zA-Z0-9:;,\[\]\(\)]

Para implementar dichos componentes sobre el analizador hay que tener en cuenta qué se debe almacenar acerca de ellos, en cuanto a los operadores almacenan el operador. Sobre los identificadores y las palabras reservadas se almacenan el valor y el número de línea. Acerca de los números se almacenan un booleano que indica si es entero o no, en caso de no serlo se trataría como real, y el valor del número. Del resto de componentes no es necesario almacenar ningún atributo.

# Descripción del MDD

Sobre el analizador léxico que hemos implementado cabe destacar que basa su ejecución en la lectura de todo el fichero carácter a carácter hasta que llega al fin de fichero, en función del carácter que encuentre tomará un estado u otro del MDD implementado.

Una de las principales categorías que debemos reconocer en nuestro analizador léxico son las palabras reservadas, tenemos que tener en cuenta que nuestro lenguaje no puede declarar identificadores de similar nombre a estas palabras de manera que nuestro autómata deberá reconocer todas estas palabras reservadas y tener en cuenta que cuando esta difiera en algún carácter se trata de un identificador, por ejemplo, supongamos la palabra “VAR” que se trata de una palabra reservada.

La MDD tendría la siguiente apariencia. V → A → R, e inicialmente encontramos una “V” si el carácter posterior fuera una “O” pasaríamos a la parte de la MDD que se encarga de devolver un identificador, sin embargo, si encontramos una “E” pasaríamos a la parte de la MDD que se encarga de reconocer la palabra reservada “VECTOR”.

Para nuestra MDD los operadores + y – corresponde a la categoría OpAdd.

Los operadores \* o / a la categoría OpMult.

Las aperturas y cierres de corchetes, y paréntesis requieren ser reconocidos por nuestro analizador léxico de manera independiente en las respectivas categorías, CorAp, CorCi, ParentAp, ParentCi, ya que en fases posteriores al analizador léxico es necesario esta categoría para reconocer el principio y cierre de las expresiones que estos contienen.

Las aperturas de llave y cierre de la misma corresponden a los comentarios que introduciremos en el programa, debemos detectar y dar parte del error cuando no se encuentre la apertura o cierre de la misma donde corresponda. Ya que los comentarios deben ser ignorados en fases posteriores al analizador lexico. De manera que cuando encontramos una llave de apertura ignoramos todo lo posterior hasta encontrar el cierre de la misma.

También poseen categorías diferentes el ;(PtoComa), la ','(Coma) y el .(Punto).

Cuando aparecen los dos puntos, en nuestro lenguaje se presentan dos casos, los : si no van precedidos de un signo = corresponden a la categoría DosPtos, en caso de ir acompañados de el símbolo = tenemos que tener en cuenta que corresponde a la operación de asignación (OpAsigna).

El símbolo igual = corresponde a la categoria de OpRel.

Para los OpRel también tenemos que tener en cuenta el caso menor, mayor y también los casos mayor igual o menor igual, además del operador distinto definido como <>, los operadores < o > que no van seguidos de un signo = o signos < y > también serán considerados OpRel.

Los números también corresponden a una categoría en nuestra MDD, tenemos que distinguir que podemos encontrarnos dos tipos de números, enteros y floats, los números serán enteros hasta que aparezca el símbolo '.' que entonces pasaran a ser float.

También contemplamos que cuando se encuentra un salto de línea /n o /r dependiendo del sistema operativo tenemos que continuar el análisis, es decir, no devolvemos ninguna categoría.

Los espacios al igual que los saltos de linea deben saltarse, es decir primeramente reconocemos un espacio y posteriormente seguimos saltando mientras nos encontramos mas espacios, hasta llegar a un carácter diferente a este.

# Descripción de los errores

Además hay que tener en cuenta que se incluye la posibilidad de insertar comentarios, así si detecta una llave de apertura “{“ ignora todo lo que haya después hasta que encuentre el cierre “}”, de no encontrarlo habría un error.

También tenemos en cuenta que si encuentra una llave de cierre de comentario sin abrir es un error, pues un comentario debe abrirse antes.

# Funciones y recursos utilizados

Para el analizador léxico a nivel de implementación es necesario importar los siguientes ficheros:

* Componentes.
* Errores.
* Flujo.
* String.
* Sys.

De la librería sys importamos argv y de la librería sets importamos InmutableSet, que se emplea para definir el conjunto de palabras reservadas, se declara como InmutableSet para dejar claro que este conjunto bajo ningún concepto se puede modificar.

La clase Analex contiene:

* La declaración del conjunto de palabras reservadas, como hemos citado anteriormente InmutableSet.
* El constructor de la clase, donde pasamos un flujo por parámetro, para facilitar el flujo de caracteres de entrada, también se declara self.nlinea para identificar la posición de los posibles errores.
* Se define también la función Analiza(self): donde se identifica los distintos componentes léxicos y como código se adjunta con la memoria, tenemos en cuenta que nuestra implementación para el analizador léxico reside en este método, donde devolvemos el correspondiente componente léxico.

En la descripción del MDD hemos realizado un resumen que afianza los conceptos que hemos seguido a la hora de la implementación de las diferentes categorías léxicas.

* En cuanto a la ejecución del hilo, disponemos del método main que se encarga de recoger el nombre del fichero pasado por parámetros, abrirlo y establecer un flujo sobre el mismo. Creando un objeto Analex correspondiente cuya entrada es el flujo del fichero.

El fichero se recorrerá carácter a carácter.

Recordamos que el hecho de recoger la información relativa al número de líneas se incrementa en el caso de \n correspondiente al salto de línea

# Analizador Sintáctico

# Tabla de análisis, condición LL(1) y errores

Lo primero que verificamos es que la gramática no es LL1, pues encontramos una parte común en la parte derecha de dos reglas:

<expresion> -> <expr\_simple> oprel <expr\_simple>

<expresion> -> <expr\_simple>

Así para solucionarlo quedaría así:

<expresion> -> <expr\_simple> <expr\_aux>

<expr\_aux> -> oprel <expr\_simple> | λ

Obteniendo la siguiente gramática:

<Programa> -> PROGRAMA id ; <decl\_var><decl\_subprog> <instrucciones> .

<decl\_var> -> VAR <lista\_id> : <tipo> ; <decl\_v> | λ

<decl\_v> -> <lista\_id> : <tipo> ; <decl\_v> | λ

<lista\_id> -> id <resto\_listaid>

<resto\_listaid> -> , <lista\_id> | λ

<tipo> -> <tipo\_std> | VECTOR [ num ] de <tipo>

<tipo\_std> -> ENTERO | REAL | BOOLEANO

<decl\_subprg> -> <decl\_sub> ; <decl\_subprg> | λ

<decl\_sub> -> PROC id ; <instrucciones> | FUNCION id : <tipo\_std> ; <instrucciones>

<instrucciones> -> INICIO <lista\_inst> FIN

<lista\_inst> -> <instruccion> ; <lista\_inst> | λ

<instruccion> -> INICIO <lista\_inst> FIN | <inst\_simple> | <inst\_e/s>

-> SI <expresion> ENTONCES <instruccion> SINO <instruccion>

-> MIENTRAS <expresion> HACER <instruccion>

<inst\_simple> -> id <resto\_instsimple>

<resto\_instsimple> -> opasigna <expresion> | [ <expr\_simple> ] opasigna <expresion> | λ

<variable> -> id <resto\_var>

<resto\_var> -> [ <expr\_simple> ] | λ

<inst\_e/s> -> LEE ( id ) | ESCRIBE ( <expr\_simple> )

<expresion> -> <expr\_simple> <expr\_aux>

<expr\_aux> -> oprel <expr\_simple> | λ

<expr\_simple> -> <termino> <resto\_exsimple> | <signo> <termino> <resto\_exsimple>

<resto\_exsimple> -> opsuma <termino> <resto\_exsimple> | O <termino> <resto\_exsimple> | λ

<termino> -> <factor> <resto\_term>

<resto\_term> -> opmult <factor> <resto\_term> | Y <factor> <resto\_term> | λ

<factor> -> <variable> | num | ( <expresion> ) | NO <factor> | CIERTO | FALSO

A continuación, hemos procedido a calcular los primeros y los siguientes:

|  |  |  |
| --- | --- | --- |
|  | Primeros | Anulable |
| <Programa> | { **PROGRAMA** } | No |
| <decl\_var> | { **VAR** } | ø | Si |
| <decl\_v> | { **id** } | ø | Si |
| <lista\_id> | { **id** } | No |
| <resto\_listaid> | { **,** } | ø | Si |
| <tipo> | { **ENTERO** , **REAL** , **BOOLEANO** } | { **VECTOR** } | No |
| <tipo\_std> | { **ENTERO** } | { **REAL** } | { **BOOLEANO** } | No |
| <decl\_subprg> | { **PROC** , **FUNCION** } | ø | Si |
| <decl\_sub> | { **PROC** } | { **FUNCION** } | No |
| <instrucciones> | { **INICIO** } | No |
| <lista\_inst> | { **INICIO** , **id** , **LEE** , **ESCRIBE** , **SI** , **MIENTRAS** } | ø | Si |
| <instruccion> | { **INICIO** } | { **id** } | { **LEE** , **ESCRIBE** } | { **SI** } | { **MIENTRAS** } | No |
| <inst\_simple> | { **id** } | No |
| <resto\_instsimple> | { **opasigna** } | { **[** } | ø | Si |
| <variable> | { **id** } | No |
| <resto\_var> | { **[** } | ø | Si |
| <inst\_e/s> | { **LEE** } | { **ESCRIBE** } | No |
| <expresion> | { **id , num , ( , NO , CIERTO , FALSO** , **+** , **-** } | No |
| <expr\_aux> | { **oprel** } | ø | Si |
| <expr\_simple> | { **id , num , ( , NO , CIERTO , FALSO** } | { **+** , **-** } | No |
| <resto\_exsimple> | { **opsuma** } | { **O** } | ø | Si |
| <termino> | { **id** , **num** , **(** , **NO** , **CIERTO , FALSO** } | No |
| <resto\_term> | { **opmult** } | { **Y** } | ø | Si |
| <factor> | { **id** } | { **num** } | { **(** } | { **NO** } | { **CIERTO** } | { **FALSO** } | No |
| <signo> | { **+** } | { **-** } | No |

|  |  |  |
| --- | --- | --- |
|  | Siguientes | Anulable |
| <Programa> | { **$** } | No |
| <decl\_var> | { **PROC** , **FUNCION** , **INICIO** } | Si |
| <decl\_v> | { **PROC** , **FUNCION** , **INICIO** } | Si |
| <lista\_id> | { **:** } | No |
| <resto\_listaid> | { **:** } | Si |
| <tipo> | { **;** } | No |
| <tipo\_std> | { **;** } | No |
| <decl\_subprg> | { **INICIO** } | Si |
| <decl\_sub> | { **;** } | No |
| <instrucciones> | { **.** , **;** } | No |
| <lista\_inst> | { **FIN** } | Si |
| <instruccion> | { **;** , **SINO** } | No |
| <inst\_simple> | { **;** , **SINO** } | No |
| <resto\_instsimple> | { **;** , **SINO** } | Si |
| <variable> | { **opmult** , **Y** , **opsuma** , **O** , **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** , **SINO** } | No |
| <resto\_var> | { **opmult** , **Y** , **opsuma** , **O** , **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** , **SINO** } | Si |
| <inst\_e/s> | { **;** , **SINO** } | No |
| <expresion> | { **ENTONCES** , **HACER** , **)** , **;** , **SINO** } | No |
| <expr\_aux> | { **ENTONCES** , **HACER** , **)** , **;** , **SINO** } | Si |
| <expr\_simple> | { **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** , **SINO** } | No |
| <resto\_exsimple> | { **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** , **SINO** } | Si |
| <termino> | { **opsuma** , **O** , **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** , **SINO** } | No |
| <resto\_term> | { **opsuma** , **O** , **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** , **SINO** } | Si |
| <factor> | { **opmult** , **Y** , **opsuma** , **O** , **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** , **SINO** } | No |
| <signo> | { **id** , **num** , **(** , **NO** , **CIERTO , FALSO** } | No |

Así de esta forma podemos hallar la tabla de análisis:

|  |  |  |
| --- | --- | --- |
|  | PROGRAMA | id |
| <Programa> | **PROGRAMA id** **;** <decl\_var> <decl\_subprg> <instrucciones> **.** |  |
| <decl\_var> |  |  |
| <decl\_v> |  | <lista\_id> **:** <tipo> **;** <decl\_v> |
| <lista\_id> |  | **id** <resto\_listaid> |
| <resto\_listaid> |  |  |
| <tipo> |  |  |
| <tipo\_std> |  |  |
| <decl\_subprg> |  |  |
| <decl\_sub> |  |  |
| <instrucciones> |  |  |
| <lista\_inst> |  | <instruccion> **;** <lista\_inst> |
| <instruccion> |  | <inst\_simple> |
| <inst\_simple> |  | **id** <resto\_instsimple> |
| <resto\_instsimple> |  |  |
| <variable> |  | **id** <resto\_var> |
| <resto\_var> |  |  |
| <inst\_e/s> |  |  |
| <expresion> |  | <expr\_simple> <expr\_aux> |
| <expr\_aux> |  |  |
| <expr\_simple> |  | <termino> <resto\_exsimple> |
| <resto\_exsimple> |  |  |
| <termino> |  | <factor> <resto\_term> |
| <resto\_term> |  |  |
| <factor> |  | <variable> |
| <signo> |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | ; | . | VAR | : | , | VECTOR |
| <Programa> |  |  |  |  |  |  |
| <decl\_var> |  |  | **VAR** <lista\_id> **:** <tipo> **;** <decl\_v> |  |  |  |
| <decl\_v> |  |  |  |  |  |  |
| <lista\_id> |  |  |  |  |  |  |
| <resto\_listaid> |  |  |  | λ | **,** <lista\_id> |  |
| <tipo> |  |  |  |  |  | **VECTOR** **[** **num** **]** **de** <tipo> |
| <tipo\_std> |  |  |  |  |  |  |
| <decl\_subprg> |  |  |  |  |  |  |
| <decl\_sub> |  |  |  |  |  |  |
| <instrucciones> |  |  |  |  |  |  |
| <lista\_inst> |  |  |  |  |  |  |
| <instruccion> |  |  |  |  |  |  |
| <inst\_simple> |  |  |  |  |  |  |
| <resto\_instsimple> | λ |  |  |  |  |  |
| <variable> |  |  |  |  |  |  |
| <resto\_var> | λ |  |  |  |  |  |
| <inst\_e/s> |  |  |  |  |  |  |
| <expresion> |  |  |  |  |  |  |
| <expr\_aux> | λ |  |  |  |  |  |
| <expr\_simple> |  |  |  |  |  |  |
| <resto\_exsimple> | λ |  |  |  |  |  |
| <termino> |  |  |  |  |  |  |
| <resto\_term> | λ |  |  |  |  |  |
| <factor> |  |  |  |  |  |  |
| <signo> |  |  |  |  |  |  |

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | [ | ] | num | de | ENTERO | REAL |
| <Programa> |  |  |  |  |  |  |
| <decl\_var> |  |  |  |  |  |  |
| <decl\_v> |  |  |  |  |  |  |
| <lista\_id> |  |  |  |  |  |  |
| <resto\_listaid> |  |  |  |  |  |  |
| <tipo> |  |  |  |  | <tipo\_std> | <tipo\_std> |
| <tipo\_std> |  |  |  |  | **ENTERO** | **REAL** |
| <decl\_subprg> |  |  |  |  |  |  |
| <decl\_sub> |  |  |  |  |  |  |
| <instrucciones> |  |  |  |  |  |  |
| <lista\_inst> |  |  |  |  |  |  |
| <instruccion> |  |  |  |  |  |  |
| <inst\_simple> |  |  |  |  |  |  |
| <resto\_instsimple> | **[** <expr\_simple> **]** **opasigna** <expresion> |  |  |  |  |  |
| <variable> |  |  |  |  |  |  |
| <resto\_var> | **[** <expr\_simple> **]** | λ |  |  |  |  |
| <inst\_e/s> |  |  |  |  |  |  |
| <expresion> |  |  | <expr\_simple> <expr\_aux> |  |  |  |
| <expr\_aux> |  |  |  |  |  |  |
| <expr\_simple> |  |  | <termino> <resto\_exsimple> |  |  |  |
| <resto\_exsimple> |  | λ |  |  |  |  |
| <termino> |  |  | <factor> <resto\_term> |  |  |  |
| <resto\_term> |  | λ |  |  |  |  |
| <factor> |  |  | **num** |  |  |  |
| <signo> |  |  |  |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | BOOLEANO | PROC | FUNCION |
| <Programa> |  |  |  |
| <decl\_var> |  | λ | λ |
| <decl\_v> |  | λ | λ |
| <lista\_id> |  |  |  |
| <resto\_listaid> |  |  |  |
| <tipo> | <tipo\_std> |  |  |
| <tipo\_std> | **BOOLEANO** |  |  |
| <decl\_subprg> |  | <decl\_sub> **;** <decl\_subprg> | <decl\_sub> **;** <decl\_subprg> |
| <decl\_sub> |  | **PROC** **id** **;** <instrucciones> | **FUNCION** **id** **:** <tipo\_std> **;** <instrucciones> |
| <instrucciones> |  |  |  |
| <lista\_inst> |  |  |  |
| <instruccion> |  |  |  |
| <inst\_simple> |  |  |  |
| <resto\_instsimple> |  |  |  |
| <variable> |  |  |  |
| <resto\_var> |  |  |  |
| <inst\_e/s> |  |  |  |
| <expresion> |  |  |  |
| <expr\_aux> |  |  |  |
| <expr\_simple> |  |  |  |
| <resto\_exsimple> |  |  |  |
| <termino> |  |  |  |
| <resto\_term> |  |  |  |
| <factor> |  |  |  |
| <signo> |  |  |  |

|  |  |  |
| --- | --- | --- |
|  | INICIO | FIN |
| <Programa> |  |  |
| <decl\_var> | λ |  |
| <decl\_v> | λ |  |
| <lista\_id> |  |  |
| <resto\_listaid> |  |  |
| <tipo> |  |  |
| <tipo\_std> |  |  |
| <decl\_subprg> | λ |  |
| <decl\_sub> |  |  |
| <instrucciones> | **INICIO** <lista\_inst> **FIN** |  |
| <lista\_inst> | <instruccion> **;** <lista\_inst> | λ |
| <instruccion> | **INICIO** <lista\_inst> **FIN** |  |
| <inst\_simple> |  |  |
| <resto\_instsimple> |  |  |
| <variable> |  |  |
| <resto\_var> |  |  |
| <inst\_e/s> |  |  |
| <expresion> |  |  |
| <expr\_aux> |  |  |
| <expr\_simple> |  |  |
| <resto\_exsimple> |  |  |
| <termino> |  |  |
| <resto\_term> |  |  |
| <factor> |  |  |
| <signo> |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | SI | ENTONCES | SINO |
| <Programa> |  |  |  |
| <decl\_var> |  |  |  |
| <decl\_v> |  |  |  |
| <lista\_id> |  |  |  |
| <resto\_listaid> |  |  |  |
| <tipo> |  |  |  |
| <tipo\_std> |  |  |  |
| <decl\_subprg> |  |  |  |
| <decl\_sub> |  |  |  |
| <instrucciones> |  |  |  |
| <lista\_inst> | <instruccion> **;** <lista\_inst> |  |  |
| <instruccion> | **SI** <expresion> **ENTONCES** <instruccion> **SINO** <instruccion> |  |  |
| <inst\_simple> |  |  |  |
| <resto\_instsimple> |  |  | λ |
| <variable> |  |  |  |
| <resto\_var> |  | λ | λ |
| <inst\_e/s> |  |  |  |
| <expresion> |  |  |  |
| <expr\_aux> |  | λ | λ |
| <expr\_simple> |  |  |  |
| <resto\_exsimple> |  | λ | λ |
| <termino> |  |  |  |
| <resto\_term> |  | λ | λ |
| <factor> |  |  |  |
| <signo> |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | MIENTRAS | HACER | opasigna |
| <Programa> |  |  |  |
| <decl\_var> |  |  |  |
| <decl\_v> |  |  |  |
| <lista\_id> |  |  |  |
| <resto\_listaid> |  |  |  |
| <tipo> |  |  |  |
| <tipo\_std> |  |  |  |
| <decl\_subprg> |  |  |  |
| <decl\_sub> |  |  |  |
| <instrucciones> |  |  |  |
| <lista\_inst> | <instruccion> **;** <lista\_inst> |  |  |
| <instruccion> | **MIENTRAS** <expresion> **HACER** <instruccion> |  |  |
| <inst\_simple> |  |  |  |
| <resto\_instsimple> |  |  | **opasigna** <expresion> |
| <variable> |  |  |  |
| <resto\_var> |  | λ |  |
| <inst\_e/s> |  |  |  |
| <expresion> |  |  |  |
| <expr\_aux> |  | λ |  |
| <expr\_simple> |  |  |  |
| <resto\_exsimple> |  | λ |  |
| <termino> |  |  |  |
| <resto\_term> |  | λ |  |
| <factor> |  |  |  |
| <signo> |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | LEE | ESCRIBE | ( |
| <Programa> |  |  |  |
| <decl\_var> |  |  |  |
| <decl\_v> |  |  |  |
| <lista\_id> |  |  |  |
| <resto\_listaid> |  |  |  |
| <tipo> |  |  |  |
| <tipo\_std> |  |  |  |
| <decl\_subprg> |  |  |  |
| <decl\_sub> |  |  |  |
| <instrucciones> |  |  |  |
| <lista\_inst> | <instruccion> **;** <lista\_inst> | <instruccion> **;** <lista\_inst> |  |
| <instruccion> | <inst\_e/s> | <inst\_e/s> |  |
| <inst\_simple> |  |  |  |
| <resto\_instsimple> |  |  |  |
| <variable> |  |  |  |
| <resto\_var> |  |  |  |
| <inst\_e/s> | **LEE** **(** **id** **)** | **ESCRIBE** **(** <expr\_simple> **)** |  |
| <expresion> |  |  | <expr\_simple> <expr\_aux> |
| <expr\_aux> |  |  |  |
| <expr\_simple> |  |  | <termino> <resto\_exsimple> |
| <resto\_exsimple> |  |  |  |
| <termino> |  |  | <factor> <resto\_term> |
| <resto\_term> |  |  |  |
| <factor> |  |  | **(** <expresion> **)** |
| <signo> |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | ) | oprel | opsuma |
| <Programa> |  |  |  |
| <decl\_var> |  |  |  |
| <decl\_v> |  |  |  |
| <lista\_id> |  |  |  |
| <resto\_listaid> |  |  |  |
| <tipo> |  |  |  |
| <tipo\_std> |  |  |  |
| <decl\_subprg> |  |  |  |
| <decl\_sub> |  |  |  |
| <instrucciones> |  |  |  |
| <lista\_inst> |  |  |  |
| <instruccion> |  |  |  |
| <inst\_simple> |  |  |  |
| <resto\_instsimple> |  |  |  |
| <variable> |  |  |  |
| <resto\_var> | λ | λ | λ |
| <inst\_e/s> |  |  |  |
| <expresion> |  |  |  |
| <expr\_aux> | λ | **oprel** <expr\_simple> |  |
| <expr\_simple> |  |  |  |
| <resto\_exsimple> | λ | λ | **opsuma** <termino> <resto\_exsimple> |
| <termino> |  |  |  |
| <resto\_term> | λ | λ | λ |
| <factor> |  |  |  |
| <signo> |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | O | opmult | Y |
| <Programa> |  |  |  |
| <decl\_var> |  |  |  |
| <decl\_v> |  |  |  |
| <lista\_id> |  |  |  |
| <resto\_listaid> |  |  |  |
| <tipo> |  |  |  |
| <tipo\_std> |  |  |  |
| <decl\_subprg> |  |  |  |
| <decl\_sub> |  |  |  |
| <instrucciones> |  |  |  |
| <lista\_inst> |  |  |  |
| <instruccion> |  |  |  |
| <inst\_simple> |  |  |  |
| <resto\_instsimple> |  |  |  |
| <variable> |  |  |  |
| <resto\_var> | λ | λ | λ |
| <inst\_e/s> |  |  |  |
| <expresion> |  |  |  |
| <expr\_aux> |  |  |  |
| <expr\_simple> |  |  |  |
| <resto\_exsimple> | **O** <termino> <resto\_exsimple> |  |  |
| <termino> |  |  |  |
| <resto\_term> | λ | **opmult** <factor> <resto\_term> | **Y** <factor> <resto\_term> |
| <factor> |  |  |  |
| <signo> |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | NO | CIERTO | FALSO |
| <Programa> |  |  |  |
| <decl\_var> |  |  |  |
| <decl\_v> |  |  |  |
| <lista\_id> |  |  |  |
| <resto\_listaid> |  |  |  |
| <tipo> |  |  |  |
| <tipo\_std> |  |  |  |
| <decl\_subprg> |  |  |  |
| <decl\_sub> |  |  |  |
| <instrucciones> |  |  |  |
| <lista\_inst> |  |  |  |
| <instruccion> |  |  |  |
| <inst\_simple> |  |  |  |
| <resto\_instsimple> |  |  |  |
| <variable> |  |  |  |
| <resto\_var> |  |  |  |
| <inst\_e/s> |  |  |  |
| <expresion> | <expr\_simple> <expr\_aux> | <expr\_simple> <expr\_aux> | <expr\_simple> <expr\_aux> |
| <expr\_aux> |  |  |  |
| <expr\_simple> | <termino> <resto\_exsimple> | <termino> <resto\_exsimple> | <termino> <resto\_exsimple> |
| <resto\_exsimple> |  |  |  |
| <termino> | <factor> <resto\_term> | <factor> <resto\_term> | <factor> <resto\_term> |
| <resto\_term> |  |  |  |
| <factor> | **NO** <factor> | **CIERTO** | **FALSO** |
| <signo> |  |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
|  | + | - | $ |
| <Programa> |  |  |  |
| <decl\_var> |  |  |  |
| <decl\_v> |  |  |  |
| <lista\_id> |  |  |  |
| <resto\_listaid> |  |  |  |
| <tipo> |  |  |  |
| <tipo\_std> |  |  |  |
| <decl\_subprg> |  |  |  |
| <decl\_sub> |  |  |  |
| <instrucciones> |  |  |  |
| <lista\_inst> |  |  |  |
| <instruccion> |  |  |  |
| <inst\_simple> |  |  |  |
| <resto\_instsimple> |  |  |  |
| <variable> |  |  |  |
| <resto\_var> |  |  |  |
| <inst\_e/s> |  |  |  |
| <expresion> | <expr\_simple> <expr\_aux> | <expr\_simple> <expr\_aux> |  |
| <expr\_aux> |  |  |  |
| <expr\_simple> | <signo> <termino> <resto\_exsimple> | <signo> <termino> <resto\_exsimple> |  |
| <resto\_exsimple> |  |  |  |
| <termino> |  |  |  |
| <resto\_term> |  |  |  |
| <factor> |  |  |  |
| <signo> | **+** | **-** |  |

Así con la tabla de análisis se puede proceder a elaborar el analizador sintáctico junto a la gestión de errores sintácticos.

Para la gestión de errores se han tenido en cuenta los siguientes de donde se tiene el error, si falla en un analyze. En caso de fallar en un check de un componente, se tienen en cuenta los siguientes de ese componente, así como los siguientes de la parte izquierda de la regla. Así elaborar el conjunto sincronización, utilizando lo que se espera encontrar.

Fragmento de código para mostrar cómo se utiliza la tabla de análisis para implementar el analizador sintáctico:

def analyzeDeclVar(self, \*\*kwargs):

        if (self.component == None):

            return

        if (self.component.cat == "PR" and self.component.valor == "VAR"):

            self.advance()

            self.analyzeListaId()

            self.check(cat="DosPtos", sync=set([None, "PR"]), spr=set(["PROC", "FUNCION", "INICIO", "ENTERO", "REAL", "BOOLEANO", "VECTOR"]))

            self.analyzeTipo()

            self.check(cat="PtoComa", sync=set([None, "PR", "Identif"]), spr=set(["PROC", "FUNCION", "INICIO"]))

            self.analyzeDeclV()

        elif (not (self.component.cat == "PR" and (self.component.valor in ["PROC", "FUNCION", "INICIO"] ))):

            self.error(msg='VAR, PROC, FUNCION, INICIO',

                sync=set([None, "PR"]),

                spr=set(["PROC", "FUNCION", "INICIO"]))

En la tabla de análisis encontramos <decl\_var> con VAR: “VAR <lista\_id> : <tipo> ; <decl\_v>”; <decl\_var> con PROC, FUNCION e INICIO: “ λ ”.

Así en el código podemos ver que se comprueba en primera instnacia si es una palabra reservada con valor VAR, posteriormente analiza <lista\_id>, comprueba si a continuación le siguen dos puntos ( : ), y se analiza <tipo>, comprobando que después le siga un punto y coma ( ; ), analizando <decl\_v> posteriormente. En caso de no encontrarse lo anterior o “ λ “ lanza un error.

Al realizar el analizador sintáctico y al realizar distintas pruebas que debería pasar detectamos un error en la gramática definida inicialmente, faltando un punto y coma antes del “SINO” en la especificación sintáctica.

<instruccion> -> SI <expresion> ENTONCES <instruccion> ; SINO <instruccion>

De esta forma queda solucionado, pero evidentemente modificaría el conjunto de siguientes.

Así habría que eliminar SINO de los siguientes, puesto que en lugar de SINO correspondería el punto y coma, y el resto lo obtiene porque lo extienden de <instruccion>.

|  |  |  |
| --- | --- | --- |
|  | Siguientes | Anulable |
| <Programa> | { **$** } | No |
| <decl\_var> | { **PROC** , **FUNCION** , **INICIO** } | Si |
| <decl\_v> | { **PROC** , **FUNCION** , **INICIO** } | Si |
| <lista\_id> | { **:** } | No |
| <resto\_listaid> | { **:** } | Si |
| <tipo> | { **;** } | No |
| <tipo\_std> | { **;** } | No |
| <decl\_subprg> | { **INICIO** } | Si |
| <decl\_sub> | { **;** } | No |
| <instrucciones> | { **.** , **;** } | No |
| <lista\_inst> | { **FIN** } | Si |
| <instruccion> | { **;** } | No |
| <inst\_simple> | { **;** } | No |
| <resto\_instsimple> | { **;** } | Si |
| <variable> | { **opmult** , **Y** , **opsuma** , **O** , **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** } | No |
| <resto\_var> | { **opmult** , **Y** , **opsuma** , **O** , **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** } | Si |
| <inst\_e/s> | { **;** , **SINO** } | No |
| <expresion> | { **ENTONCES** , **HACER** , **)** , **;** } | No |
| <expr\_aux> | { **ENTONCES** , **HACER** , **)** , **;** } | Si |
| <expr\_simple> | { **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** } | No |
| <resto\_exsimple> | { **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** } | Si |
| <termino> | { **opsuma** , **O** , **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** } | No |
| <resto\_term> | { **opsuma** , **O** , **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** } | Si |
| <factor> | { **opmult** , **Y** , **opsuma** , **O** , **]** , **)** , **oprel** , **ENTONCES** , **HACER** , **;** } | No |
| <signo> | { **id** , **num** , **(** , **NO** , **CIERTO , FALSO** } | No |

En la tabla de análisis cambiaría lo siguiente:

|  |  |  |
| --- | --- | --- |
|  | SI | SINO |
| <Programa> |  |  |
| <decl\_var> |  |  |
| <decl\_v> |  |  |
| <lista\_id> |  |  |
| <resto\_listaid> |  |  |
| <tipo> |  |  |
| <tipo\_std> |  |  |
| <decl\_subprg> |  |  |
| <decl\_sub> |  |  |
| <instrucciones> |  |  |
| <lista\_inst> | <instruccion> **;** <lista\_inst> |  |
| <instruccion> | **SI** <expresion> **ENTONCES** <instruccion> ; **SINO** <instruccion> |  |
| <inst\_simple> |  |  |
| <resto\_instsimple> |  |  |
| <variable> |  |  |
| <resto\_var> |  |  |
| <inst\_e/s> |  |  |
| <expresion> |  |  |
| <expr\_aux> |  |  |
| <expr\_simple> |  |  |
| <resto\_exsimple> |  |  |
| <termino> |  |  |
| <resto\_term> |  |  |
| <factor> |  |  |
| <signo> |  |  |

Cabe destacar que en analyzeInstrucción se añade la comprobación del punto y coma antes del SINO cuando detecta un SI:

elif (self.component.cat == "PR" and self.component.valor == "SI"):

            self.advance()

            self.analyzeExpresion()

            self.check(cat="PR", valor="ENTONCES", sync=set([None, "Identif", "PtoComa", "PR"]), spr=set(["SINO", "INICIO", "LEE", "ESCRIBE", "SI", "MIENTRAS"]))

            self.analyzeInstruccion()

            self.check(cat="PtoComa", sync=set([None, "PtoComa", "PR"]), spr=set(["SINO"]))

            self.check(cat="PR", valor="SINO", sync=set([None, "Identif", "PtoComa", "PR"]), spr=set(["SINO", "INICIO", "LEE", "ESCRIBE", "SI", "MIENTRAS"]))

            self.analyzeInstruccion()

Así también habría que actualizar los conjuntos de sincronización cambiando el SINO en los que estaba antes.

# Funciones utilizadas

* advance: avanza un lugar, contemplando errores o fin de ejecución.
* sincroniza: pide al analizador léxico el siguiente componente hasta encontrar uno que esté encontrar uno que esté en el conjunto de sincronización.
* error: muestra un error sintáctico encontrado y el elemento esperado.

Se utiliza también el flujo, como en el analizador léxico, y además se llama desde este al analizador léxico en el método main.

# Analizador Semántico

# Comprobación de identificadores

Para implementar la parte del analizador semántico que trata acerca de comprobar la aparición de identificadores repetidos se han añadido las siguientes reglas a la gramática:

<Programa> -> PROGRAMA id ; {<decl\_var>.ids = [id.valor]} <decl\_var> {<decl\_subprg>.ids = <decl\_var>.ids} <decl\_subprg> {<instrucciones>.ids = <decl\_subprg>.ids} <instrucciones> .

<decl\_var> -> VAR {<lista\_id>.ids = <decl\_var>.ids} <lista\_id> : <tipo> ; {<decl\_v>.ids = <lista\_id>.ids} <decl\_v> {<decl\_var>.ids = <decl\_v>.ids}

-> λ

<decl\_v> -> {<lista\_id>.ids = <decl\_v>.ids} <lista\_id> : <tipo> ; {<decl\_v>1.ids = <lista\_id>.ids} <decl\_v>1 {<decl\_v>.ids = <decl\_v>1.ids}

-> λ

<lista\_id> -> id {if id.valor in <lista\_id>.ids then ERROR(identif repetido);

else <lista\_id>.ids.append(id.valor);

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

<resto\_listaid> -> , {<lista\_id>.ids = <resto\_listaid>.ids} <lista\_id> {<resto\_listaid>.ids = <lista\_id>.ids}

-> λ

<tipo> -> <tipo\_std> | VECTOR [ num ] de <tipo>

<tipo\_std> -> ENTERO | REAL | BOOLEANO

<decl\_subprg> -> {<decl\_sub>.ids = <decl\_subprg>.ids} <decl\_sub> ; {<decl\_subprg>1.ids = <decl\_sub>.ids} <decl\_subprg>1 {<decl\_subprg>.ids = <decl\_subprg>1.ids}

-> λ

<decl\_sub> -> PROC id {if id.valor in <decl\_sub>.ids then ERROR(identif repetido);

<decl\_sub>.ids.append(id.valor);} ; {<instrucciones>.ids = <decl\_sub>.ids} <instrucciones>

-> FUNCION id {if id.valor in <decl\_sub>.ids then ERROR(identif repetido);

<decl\_sub>.ids.append(id.valor);} : <tipo\_std> ; {<instrucciones>.ids = <decl\_sub>.ids} <instrucciones>

<instrucciones> -> INICIO {<lista\_inst>.ids = <instrucciones>.ids} <lista\_inst> FIN

<lista\_inst> -> {<instruccion>.ids = <lista\_inst>.ids} <instruccion> ; {<lista\_inst>1.ids = <lista\_inst>.ids} <lista\_inst>1 | λ

<instruccion> -> INICIO {<lista\_inst>.ids = <instruccion>.ids} <lista\_inst> FIN

-> {<inst\_simple>.ids = <instruccion>.ids} <inst\_simple>

-> {<inst\_e/s>.ids = <instruccion>.ids} <inst\_e/s>

-> SI {<expresion>.ids = <instruccion>.ids} <expresion> ENTONCES {<instruccion>1.ids = <instruccion>.ids} <instruccion>1 ; SINO {<instruccion>2.ids = <instruccion>.ids} <instruccion>2

-> MIENTRAS {<expresion>.ids = <instruccion>.ids} <expresion> HACER {<instruccion>1.ids = <instruccion>.ids} <instruccion>1

<inst\_simple> -> id {if id.valor not in <inst\_simple>.ids then ERROR(Variable no declarada)} {<resto\_instsimple>.ids = <inst\_simple>.ids} <resto\_instsimple>

<resto\_instsimple> -> opasigna {<expresion>.ids = <resto\_instsimple>.ids} <expresion> | [ {<expr\_simple>.ids = <resto\_instsimple>.ids} <expr\_simple> ] opasigna {<expresion>.ids = <resto\_instsimple>.ids} <expresion> | λ

<variable> -> {if id.valor not in <inst\_simple>.ids then ERROR(Variable no declarada)} id {<resto\_var>.ids = <variable>.ids} <resto\_var>

<resto\_var> -> [ {<expr\_simple>.ids = <resto\_var>.ids} <expr\_simple> ] | λ

<inst\_e/s> -> LEE ( id {if id.valor not in <inst\_simple>.ids then ERROR(Variable no declarada)} ) | ESCRIBE ( {<expr\_simple>.ids = <resto\_var>.ids} <expr\_simple> )

<expresion> -> {<expr\_simple>.ids = <expresion>.ids} <expr\_simple> {<expr\_aux>.ids = <expresion>.ids} <expr\_aux>

<expr\_aux> -> oprel {<expr\_simple>.ids = <expr\_aux>.ids} <expr\_simple> | λ

<expr\_simple> -> {<termino>.ids = <expr\_simple>.ids} <termino> {<resto\_exsimple>.ids = <expr\_simple>.ids} <resto\_exsimple>

-> <signo> {<termino>.ids = <expr\_simple>.ids} <termino> {<resto\_exsimple>.ids = <expr\_simple>.ids} <resto\_exsimple>

<resto\_exsimple> -> opsuma {<termino>.ids = <resto\_exsimple>.ids} <termino> {<resto\_exsimple>1.ids = <resto\_exsimple>.ids} <resto\_exsimple>1

-> O {<termino>.ids = <resto\_exsimple>.ids} <termino> {<resto\_exsimple>1.ids = <resto\_exsimple>.ids} <resto\_exsimple>1 | λ

<termino> -> {<factor>.ids = <termino>.ids} <factor> {<resto\_term>.ids = <termino>.ids} <resto\_term>

<resto\_term> -> opmult {<factor>.ids = <resto\_term>.ids} <factor> {<resto\_term>1.ids = <resto\_term>.ids} <resto\_term>1

-> Y {<factor>.ids = <resto\_term>.ids} <factor> {<resto\_term>1.ids = <resto\_term>.ids} <resto\_term>1 | λ

<factor> -> {<variable>.ids = <factor>.ids} <variable> | num | ( {<expresion>.ids = <factor>.ids} <expresion> ) | NO {<factor>1.ids = <factor>.ids} <factor>1 | CIERTO | FALSO

<signo> -> + | -

De esta forma se implementa el analizador semántico sobre el analizador sintáctico, añadiendo los identificadores encontrados a una lista, pasándolo por parámetro y devolviéndolo en las funciones requeridas.

En caso de encontrar un identificador repetido muestra un error por cada identificador que se repite.

Fragmento de código de ejemplo:

def analyzeListaId(self, \*\*kwargs):

        lista\_id = kwargs

        if (self.component == None):

            pass

        elif (self.component.cat == "Identif"):

            v = self.component.valor

            if(v not in lista\_id['ids']):

                lista\_id['ids'].append(v)

            else:

                self.errorS(id = v)

            self.advance()

            resto\_listaid = self.analyzeRestoListaId(ids = lista\_id['ids'])

            lista\_id['ids'] = resto\_listaid['ids']

        else:

            self.error(msg='Identif',

                sync=set([None, "DosPtos"]))

        return lista\_id

Corresponde a:

<lista\_id> -> id {if id.valor in <lista\_id>.ids then ERROR(identif repetido);

else <lista\_id>.ids.append(id.valor);

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

A las funciones los parámetros se les pasan por “key arguments”, utilizando la key “ids”. Como se puede observar, si se encuentra un identificador, comprueba si está el valor en la lista de identificadores, en caso de no estar añade el identificador y continúa su ejecución, posteriormente se pasa la lista de ids a analyzeRestoListaId, que a su vez devuelve la lista, siendo este último el valor de la lista que devuelve analyzeListaId. En caso de estar presente en la lista muestra que ese identificador está repetido y no se vuelve a añadir.

También se ha implementado la comprobación de que una variable que se utiliza en el cuerpo del programa debe estar declarada previamente.

Ejemplo mediante un fragmento de código de la función analyzeVariable:

def analyzeVariable(self, \*\*kwargs):

        variable = kwargs

        if (self.component == None):

            pass

        elif (self.component.cat == "Identif"):

            if (self.component.valor not in variable['ids']):

                self.errorBefore(id = self.component.valor)

            self.advance()

            self.analyzeRestoVar(ids = variable['ids'])

        else:

            self.error(msg='Identif',

                sync=set([None, "PR", "OpMult", "OpAdd", "OpRel", "CorCi", "ParentCi", "PtoComa"]),

                spr=set(["Y", "O", "ENTONCES", "HACER"]))

Lo que devuelven las funciones, como se puede observar, son diccionarios, con el fin de facilitar el retorno de diferentes recursos y su respectivo tratamiento.

Corresponde a:

<variable> -> {if id.valor not in <inst\_simple>.ids then ERROR(Variable no declarada)} id {<resto\_var>.ids = <variable>.ids} <resto\_var>

# Comprobación de tipos

También se ha implementado la comprobación de tipos, añadiendo las siguientes reglas a la gramática:

<Programa> -> PROGRAMA id ; <decl\_var> {<decl\_subprg>.tipo\_id = <decl\_var>.tipo\_id}

<decl\_subprg> {<instrucciones>.tipo\_id = <decl\_subprg>.tipo\_id;}

<instrucciones> .

<decl\_var> -> VAR <lista\_id> : <tipo> ; <decl\_v> {<decl\_var>.tipo\_id = <decl\_v>.tipo\_id;

for id in <lista\_id>.mids:

<decl\_var>.tipo\_id[id] = <tipo>.tipo;

}

-> λ

<decl\_v> -> <lista\_id> : <tipo> ; <decl\_v>1 {<decl\_v>.tipo\_id = <decl\_v>1.tipo\_id;

for id in <lista\_id>.mids:

<decl\_v>.tipo\_id[id] = <tipo>.tipo;

}

-> λ

<lista\_id> -> id <resto\_listaid> {<lista\_id>.mids = <resto\_listaid>.mids; <lista\_id>.mids.append(id.valor)}

<resto\_listaid> -> , <lista\_id> {<resto\_listaid>.mids = <lista\_id>.mids}

-> λ

<tipo> -> <tipo\_std> {<tipo>.tipo = <tipo\_std>.tipo}

-> VECTOR [ num ] de <tipo>1 {<tipo>.tipo = (4, <tipo>1.tipo)}

<tipo\_std> -> ENTERO {<tipo\_std>.tipo = (0)}

-> REAL {<tipo\_std>.tipo = (1)}

-> BOOLEANO {<tipo\_std>.tipo = (2)}

<decl\_subprg> -> {<decl\_sub>.tipo\_id = <decl\_subprg>.tipo\_id}

<decl\_sub> ; <decl\_subprg>1 {<decl\_subprg>.tipo\_id = <decl\_subprg>1.tipo\_id;

<decl\_subprg>.tipo\_id[<decl\_sub>.id] = <decl\_sub>.tipo}

-> λ

<decl\_sub> -> PROC id {<decl\_sub>.id = id.valor; <decl\_sub>.tipo = (5); <decl\_sub>.tipo\_id[id.valor] = (5)}

; {<instrucciones>.tipo\_id = <decl\_sub>.tipo\_id} <instrucciones>

-> FUNCION id : <tipo\_std> {<decl\_sub>.id = id.valor; <decl\_sub>.tipo = (6, <tipo\_std>.tipo); <decl\_sub>.tipo\_id[id.valor] = (6, <tipo\_std>.tipo)}

; {<instrucciones>.tipo\_id = <decl\_sub>.tipo\_id} <instrucciones>

<instrucciones> -> INICIO {<lista\_inst>.tipo\_id = <instrucciones>.tipo\_id} <lista\_inst> FIN

<lista\_inst> -> {<instruccion>.tipo\_id = <lista\_inst>.tipo\_id} <instruccion> ; {<lista\_inst>1.tipo\_id = <lista\_inst>.tipo\_id} <lista\_inst>1

-> λ

<instruccion> -> INICIO {<lista\_inst>.tipo\_id = <instruccion>.tipo\_id} <lista\_inst> FIN

-> {<inst\_simple>.tipo\_id = <instruccion>.tipo\_id} <inst\_simple>

-> {<inst\_e/s>.tipo\_id = <instruccion>.tipo\_id} <inst\_e/s>

-> SI {<expresion>.tipo\_id = <instruccion>.tipo\_id} <expresion>

ENTONCES {<instruccion>1.tipo\_id = <instruccion>.tipo\_id} <instruccion>1 ;

SINO {<instruccion>2.tipo\_id = <instruccion>.tipo\_id} <instruccion>2

-> MIENTRAS {<expresion>.tipo\_id = <instruccion>.tipo\_id} <expresion>

HACER {<instruccion>1.tipo\_id = <instruccion>.tipo\_id} <instruccion>1

<inst\_simple> -> id {<resto\_instsimple>.tipo = <inst\_simple>.tipo\_id[id.valor];

<resto\_instsimple>.id = id.valor

<resto\_instsimple>.tipo\_id = <inst\_simple>.tipo\_id} <resto\_instsimple>

<resto\_instsimple> -> opasigna {<expresion>.tipo\_id = <resto\_instsimple>.tipo\_id} <expresion>

{if <expresion>.tipo != <resto\_instsimple>.tipo and !(<resto\_instsimple>.tipo[0] == 1 and <expresion>.tipo[0] == 0)

then ERROR(No se puede pasar de <expresion>.tipo a <resto\_instsimple>.tipo)}

-> {if <resto\_instsimple>.tipo[0] != 4 then ERROR(No es un VECTOR)}

[ { <expr\_simple>.tipo\_id = <resto\_instsimple>.tipo\_id } <expr\_simple>

{ if <expr\_simple>.tipo[0] != 0 then ERROR(Se debe usar un ENTERO en los accesos a VECTOR) }

] opasigna {<expresion>.tipo\_id = <resto\_instsimple>.tipo\_id}

<expresion> {if <expresion>.tipo != <resto\_instsimple>.tipo and !(<resto\_instsimple>.tipo[1][0] == 1 and <expresion>.tipo[0] == 0)

then ERROR(No se puede pasar de <expresion>.tipo a <resto\_instsimple>.tipo)}

-> λ

<variable> -> id <resto\_var> {if <resto\_var>.vector == 0

then <variable>.tipo = <variable>.tipo\_id[id.valor]

else if <variable>.tipo\_id[id.valor][0] == 4

then <variable>.tipo = <variable>.tipo\_id[id.valor][1]

else ERROR(NO ES UN VECTOR)}

<resto\_var> -> [ <expr\_simple> ]

-> λ

<inst\_e/s> -> LEE ( id )

-> ESCRIBE ( {<expr\_simple>.tipo\_id = <inst\_e/s>.tipo\_id} <expr\_simple> )

<expresion> -> {<expr\_simple>.tipo\_id = <expresion>.tipo\_id} <expr\_simple>

{<expr\_aux>.tipo = <expr\_simple>.tipo;

<expr\_aux>.tipo\_id = <expresion>.tipo\_id} <expr\_aux>

{<expresion>.tipo = <expr\_aux>.tipo}

<expr\_aux> -> oprel {<expr\_simple>.tipo\_id = <expr\_aux>.tipo\_id}

<expr\_simple> {if <expr\_aux>.tipo[0] <= 1 && <expr\_simple>.tipo[0] <= 1

then <expr\_aux>.tipo = (2)

else ERROR(<expr\_aux>.tipo and <expr\_simple>.tipo are not comparable)}

-> λ

<expr\_simple> -> {<termino>.tipo\_id = <expr\_simple>.tipo\_id} <termino>

{<resto\_exsimple>.tipo\_id = <expr\_simple>.tipo\_id} <resto\_exsimple>

{if <termino>.tipo[0] <= 1 && <resto\_exsimple>.tipo[0] <= 1

then <expr\_simple>.tipo = (max(<termino>.tipo[0], <resto\_exsimple>.tipo[0]))

else if <termino>.tipo == <resto\_exsimple>.tipo

then <expr\_simple>.tipo = <termino>.tipo

else ERROR(<termino>.tipo and <resto\_exsimple>.tipo are not compatible)}

-> <signo> {<termino>.tipo\_id = <expr\_simple>.tipo\_id} <termino>

{<resto\_exsimple>.tipo\_id = <expr\_simple>.tipo\_id} <resto\_exsimple>

{if <termino>.tipo[0] <= 1 && <resto\_exsimple>.tipo[0] <= 1

then <expr\_simple>.tipo = (max(<termino>.tipo[0], <resto\_exsimple>.tipo[0]))

else if <termino>.tipo == <resto\_exsimple>.tipo

then <expr\_simple>.tipo = <termino>.tipo

else ERROR(<termino>.tipo and <resto\_exsimple>.tipo are not compatible)}

<resto\_exsimple> -> opsuma {<termino>.tipo\_id = <resto\_exsimple>.tipo\_id} <termino>

{<resto\_exsimple>1.tipo\_id = <resto\_exsimple>.tipo\_id} <resto\_exsimple>1

{if <termino>.tipo[0] <= 1 && <resto\_exsimple>1.tipo[0] <= 1

then <resto\_exsimple>.tipo = (max(<termino>.tipo[0], <resto\_exsimple>1.tipo[0]))

else ERROR(<termino>.tipo and <resto\_exsimple>1.tipo are not compatible)}

-> O {<termino>.tipo\_id = <resto\_exsimple>.tipo\_id} <termino>

{<resto\_exsimple>1.tipo\_id = <resto\_exsimple>.tipo\_id} <resto\_exsimple>1

{if <termino>.tipo[0] == 2 && <resto\_exsimple>1.tipo[0] == 2

then <resto\_exsimple>.tipo = (2)

else ERROR(<termino>.tipo and <resto\_exsimple>1.tipo are not compatible)}

-> λ

<termino> -> {<factor>.tipo\_id = <termino>.tipo\_id} <factor>

{<resto\_term>.tipo\_id = <termino>.tipo\_id} <resto\_term>

{if <factor>.tipo[0] <= 1 && <resto\_term>.tipo[0] <= 1

then <termino>.tipo = (max(<factor>.tipo[0], <resto\_term>.tipo[0]))

else if <factor>.tipo == <resto\_term>.tipo

then <termino>.tipo = <factor>.tipo

else ERROR(<factor>.tipo and <resto\_term>.tipo are not compatible)}

<resto\_term> -> opmult {<factor>.tipo\_id = <resto\_term>.tipo\_id} <factor>

{<resto\_term>1.tipo\_id = <resto\_term>.tipo\_id} <resto\_term>1

{if <factor>.tipo[0] <= 1 && <resto\_term>1.tipo[0] <= 1

then <resto\_term>.tipo = (max(<factor>.tipo[0], <resto\_term>1.tipo[0]))

else ERROR(<factor>.tipo and <resto\_term>1.tipo are not compatible)}

-> Y {<factor>.tipo\_id = <resto\_term>.tipo\_id} <factor>

{<resto\_term>1.tipo\_id = <resto\_term>.tipo\_id} <resto\_term>1

{if <factor>.tipo[0] == 2 && <resto\_term>1.tipo[0] == 2

then <resto\_term>.tipo = (max(<factor>.tipo[0], <resto\_term>1.tipo[0]))

else ERROR(<factor>.tipo and <resto\_term>1.tipo are not compatible)}

-> λ

<factor> -> {<variable>.tipo\_id = <factor>.tipo\_id} <variable> {<factor>.tipo = <variable>.tipo}

-> num {if num.isInt == 0 then <factor>.tipo = (1) else <factor>.tipo = (0)}

-> ( {<expresion>.tipo\_id = <factor>.tipo\_id} <expresion> ) {<factor>.tipo = <expresion>.tipo}

-> NO {<factor>1.tipo\_id = <factor>.tipo\_id}

<factor>1 {<factor>.tipo = (2); if <factor>1.tipo != (2) then ERROR(NO BOOLEANO)}

-> CIERTO

-> FALSO

<signo> -> + | -

Fragmento de analyzeRestoInstSimple:

def analyzeRestoInstSimple(self, \*\*kwargs):

        resto\_instsimple = kwargs

        if (self.component == None):

            pass

        elif (self.component.cat == "CorAp"):

            if resto\_instsimple['tipo'][0] != 4:

                self.errorNotVector(id = resto\_instsimple['id'])

            self.advance()

En caso de no corresponder el tipo lanza el error.

# Árbol AST

Para implementar el árbol AST se han contemplado los tipos de nodo que puede haber, así definidos en el fichero ast.py, por ejemplo como muestra el siguiente fragmento de código:

class NodoSi:

    def \_\_init\_\_(self, cond, sentenciassi, sentenciasno):

        self.cond = cond

        self.sentenciassi = sentenciassi

        self.sentenciasno = sentenciasno

class NodoSCompuesta:

    def \_\_init\_\_(self, sentencias):

        self.sentencias = sentencias

class NodoCompara:

    def \_\_init\_\_(self, op, tipo, iz, dcha):

        self.op = op

        self.tipo = tipo

        self.iz = iz

        self.dcha = dcha

Así se utiliza ast.py desde anasyn.py, generado en el método analyzePrograma con los diferentes nodos.

Fragmentos de código:

def analyzePrograma(self):

        tree = None

        if (self.component == None):

            self.errored = True

        elif (self.component.cat == "PR" and self.component.valor == "PROGRAMA"):

            self.advance()

            ids = []

            v = None

            if (hasattr(self, 'component') and hasattr(self.component, 'cat') and self.component.cat == "Identif"):

                ids = [self.component.valor]

                v = self.component.valor

            self.check(cat="Identif", sync=set([None, "PtoComa"]))

            self.check(cat="PtoComa", sync=set([None, "PR"]), spr=set(["VAR", "PROC", "FUNCION", "INICIO"]))

            decl\_var = self.analyzeDeclVar(ids = ids)

            decl\_subprg = self.analyzeDeclSubprg(ids = decl\_var['ids'], tipo\_id = decl\_var['tipo\_id'])

            instrucciones = self.analyzeInstrucciones(ids = decl\_subprg['ids'], tipo\_id = decl\_subprg['tipo\_id'])

            self.check(cat="Punto", sync=set([None]), endEx=True)

            tree = ast.AST(v, instrucciones['nodos'])

        else:

            self.error(msg='PROGRAMA',

                sync=set([None]))

        return (tree, not self.errored)

def analyzeInstrucciones(self, \*\*kwargs):

        instrucciones = kwargs

        if (self.component == None):

            pass

        elif (self.component.cat == "PR" and self.component.valor == "INICIO"):

            self.advance()

            lista\_inst = self.analyzeListaInst(ids = instrucciones['ids'], tipo\_id = instrucciones['tipo\_id'])

            self.check(cat="PR", valor="FIN", sync=set([None, "Punto", "PtoComa"]))

            instrucciones['nodos'] = lista\_inst['nodos']

...

Sobre las funciones utilizadas encontramos:

* errorS: muestra un error semántico encontrado con respecto a la repetición de variables declaradas. Modifica el atributo errored a True.
* errorBefore: muestra un error semántico con respecto a la no declaración previa de variables que se intentan utilizar. Modifica el atributo errored a True.
* error: conforme a la implementación realizada en la parte del analizador sintáctico se ha añadido que asigne a errored el valor True.
* Funciones utilizadas por el sintáctico ya que se está realizando la implementación del semántico sobre el sintáctico.
* Funciones de error de tipo que hemos definido según el error obtenido al comprobar que no concuerdan los tipos utilizados al tratar con los datos y las variables:
  + errorCast
  + errorNotVector
  + errorAVector
  + errorComp
  + errorCompat
  + errorUnsigned

Si al terminar errored es True muestra “THE PROGRAM HAS ERRORS!”. Si es false la ejecución se ha llevado a cabo correctamente, mostrando “NOICE!”.

# Conclusiones

**Descripción de las principales dificultades encontradas en la realización de la práctica.**

Las principales dificultades se encuentran al principio, hasta que se entiende el manejo completo de las clases, sus métodos y la interacción entre clases, a raíz de esto nos surgió un error que nos acompañó hasta el final del analizador sintáctico y es que realizamos una serie de modificaciones sobre el fichero flujo erróneas, las cuales posteriormente reconocimos y reparamos.

Otra de las dificultades fue la sincronización de los errores y por supuesto la incorporación del analizador semántico al sintáctico, ya que el hecho de no disponer de muchos ejemplos nos dio un añadido de dificultad, los esqueletos fueron de suma utilidad.

En cuanto a la implementación nos hemos centrado en lo más básico, no hemos empleado el uso de subrutinas en exceso para ver en todo momento el flujo y la situación en la que nos encontrábamos de manera fácil y sencilla, así encontrábamos rápidamente los errores y quizás en eso hemos podido ahorrar algo de tiempo.

Podríamos haber aplicado más técnicas de programación a alto nivel, pero el tiempo no nos lo permitía.

**Tiempo dedicado a la misma.**

Hemos dedicado todas las sesiones de prácticas, además tiempo después de clase de prácticas y algunas mañanas, en cuanto este no hemos seguido un conteo fijo de horas, pero sí que quizás en alguna semana puede haberse excedido el tiempo de horas establecido para la asignatura. Y todo esto teniendo en cuenta que no hemos empleado excesiva cantidad de tiempo en resolver los errores. La memoria también ha requerido de un gran tiempo, ya que el lenguaje es sustancioso en cuanto a requerimientos en todas las fases de análisis, por ejemplo, no hemos dibujado la MDD porque entre la representación de palabras reservadas y las demás categorías su dimensión era bastante considerable.

**Opinión sobre la práctica, aspectos positivos y aspectos negativos.**

Los aspectos han sido siempre positivos, el único impedimento el tiempo, quizás el cuatrimestre de cuarto está muy cargado de actividades, algunas de ellas sin importancia y en el momento menos adecuado. que hace que no podamos disfrutar de trabajos buenos y útiles, como por ejemplo este.

La razón por la que opinamos que es un buen trabajo es simple, recoge todos los conceptos de la asignatura y nos hace ver la importancia de cada uno de estos y su aplicación sobre los compiladores. Si la teoría en ocasiones es algo espesa, con la programación hemos conseguido observar que tiene su justificación y su gran importancia. Seguramente no queramos ser diseñadores de compiladores, pero la experiencia ha sido gratificante y útil.

Además, la atención, tutorías y el tiempo pasado en prácticas ha sido ameno y siempre muy a gusto. Así que ¡¡Chapeau!!

**Sugerencias de mejora para próximos cursos.**

Como sugerencia a la mejora de la practica pensamos que es difícil determinar una, quizás reducir un poco el lenguaje, pero puede ser un problema, ya que recoger todos los conceptos, que sean comprensibles y programables requiere de una práctica como esta.

Los compiladores son mucho más complejos, pero nos ha hecho conscientes de la dificultad que implica su implementación y la cantidad de recursos formales que estos necesitan.