FACULTAD DE INGENIERÍA INFORMÁTICA

PROCESADORES DE LENGUAJES

Práctica 3: Desarrollo de un compilador sencillo - Memoria

Realizado por:
Guillermo Tomás Fernández Martín, Luís González Naharro
{GuillermoTomas.Fernandez, Luis.Gonzalez9}@alu.uclm.es

Índice

1. Introducción **1**

2. Descripción formal del lenguaje **1**
2.1. Categorías léxicas del lenguaje 1
2.2. Máquina Discriminadora Determinista 3
2.2.1. Símbolos 3
2.2.2. Palabras 3
2.2.3. Números 3
2.3. Comprobación de la gramática 3

3. Detalles de implementación **6**
3.1. Analizador léxico 7
3.2. Analizador sintáctico 10
3.3. Analizador semántico 12
3.3.1. Tabla de símbolos 12
3.3.2. Comprobaciones semánticas 14

4. Conclusiones **16**

Listings

1. Omisión de espacios 7
2. Tratamiento de comentarios 7
3. Detección de palabras 8
4. Detección de números 9
5. Caracteres no visibles 9
6. Errores y fin de fichero 9
7. analizaRestoTerm() 11
8. analizaDeclVar() 13
9. analizaListaId() y analizaRestoListaId() 13
10. analizaInstSimple() 14
11. analizaInstES() 14
12. NodoAritmetico(AST) 16

1. Introducción

A lo largo de esta práctica, se ha realizado un compilador para un lenguaje tipo Pascal, del cual se nos proporciona la gramática que lo genera, así como ciertas pautas para la creación de este compilador. El objetivo es realizar un análisis completo hasta el punto de generación de código, el cual originalmente era orientado a la máquina virtual ROSSI. Sin embargo, debido a la falta de máquina virtual, la práctica finalmente se detiene en la creación del Árbol de Sintaxis Abstracta, sin que haya generación de código.

Esta memoria detallará las distintas partes que componen al compilador en su estado actual, de manera independiente, y explicando las partes del código que sean necesarias. Además, el proyecto con todo el código fuente, escrito en Python, se entregará de manera conjunta a esta memoria.

2. Descripción formal del lenguaje

2.1. Categorías léxicas del lenguaje

En nuestro compilador, se ha optado por representar cada categoría léxica con una clase. Por ello, al explicar cada categoría se hará referencia a la clase que la representa en el código. Todas estas clases se encuentran en el fichero *componentes.py*

- **Operador de asignación (*OpAsigna*):** el operador de asignación se representa con dos puntos y un símbolo de igual (**:=**)
- **Llave de Apertura (*LlaveAp*):** La llave de apertura es el símbolo *{*
- **Llave de Cierre (*Llaveci*):** La Llave de cierre es el símbolo *}*
- **Paréntesis de Apertura (*ParentAp*):** El paréntesis de apertura es el símbolo *(*
- **Paréntesis de Cierre (*ParentCi*):** El paréntesis de cierre es el símbolo *)*
- **Corchete de Apertura (*CorAp*):** El corchete de apertura es el símbolo *[*
- **Corchete de Cierre (*CorCi*):** El corchete de cierre es el símbolo *]*

- **Punto (*Punto*):** El punto es el símbolo *.*
- **Coma (*Coma*):** La coma es el símbolo *,*
- **Punto y Coma (*PtoComa*):** El punto y coma es el símbolo *;*
- **Dos Puntos (*DosPtos*):** Los dos puntos es el símbolo *:*
- **Fin de fichero (*EOF*):** Esta categoría léxica, que no se representa con ningún caracter, nos permite saber cuando el fichero de entrada ha terminado de ser leído.
- **Operador de Suma (*OpAdd*):** Este operador puede tomar dos valores aritméticos (*+,* -).
- **Operador de Multiplicación (*OpMult*):** Este operador puede tomar dos valores aritméticos (**, /*)
- **Operador Relacional (*OpRel*):** Este operador puede tomar los siguientes valores: $=, <>, <, <=, >=, >$
- **Identificadores (*Identif*):** Esta categoría léxica representa los identificadores que pueden aparecer en nuestro sistema. Éstos se generan a partir de la siguiente gramática:

$$\begin{aligned} letra &\rightarrow [a-zA-Z] \\ digito &\rightarrow [0-9] \\ id &\rightarrow letra(letra|digito)^* \end{aligned}$$

- **Número (*Número*):** Esta categoría léxica representa los números que pueden aparecer en nuestro sistema, ya sean enteros o reales. Éstos se generan a partir de la siguiente gramática:

$$\begin{aligned} digitos &\rightarrow digito\ digito^* \\ fraccion\_opt &\rightarrow .\ digitos \mid \lambda \\ num &\rightarrow digitos\ fraccion\_opt \end{aligned}$$

- **Palabra Reservada (*PR*):** Esta categoría léxica representa el conjunto finito de palabras reservadas de nuestro sistema. Dichas palabras reservadas son: *PROGRAMA, VAR, VECTOR, ENTERO, REAL, BOOLEANO, INICIO, FIN, SI, ENTONCES, SINO, MIENTRAS, HACER, LEE, ESCRIBE, Y, O, NO, CIERTO* y *FALSO*. Todas ellas deberán ir en mayúsculas.

2.2. Máquina Discriminadora Determinista

En esta sección se explicará cómo funciona la MDD en la implementación de nuestro compilador. Para ello, vamos a clasificar nuestras categorías léxicas en tres grandes grupos: Símbolos, Palabras y Números.

2.2.1. Símbolos

Dentro de este grupo, consideraremos todos los símbolos unitarios, los operadores y el fin de fichero. En este grupo, la mayoría de los elementos son únicos, y por lo tanto la MDD puede detectar directamente a que categoría pertenecen. En el caso de los símbolos compuestos de dos caracteres (operador de asignación u operador relacional), leemos el siguiente caracter. En caso de que sea un espacio, no existe ambiguedad (en el primer caso, sería la categoría **Dos puntos**, por ejemplo). Si no, el segundo caracter nos elimina también la ambiguedad del valor que puede tomar el operador.

2.2.2. Palabras

En este grupo consideramos los identificadores y las palabras reservadas, ya que se leen igual. La MDD lee el elemento mientras cumpla con la gramática establecida anteriormente, y una vez finaliza busca en la tabla de Palabras Reservadas si coincide con algún elemento. En caso contrario, sabemos que es un identificador.

2.2.3. Números

Si el primer caracter que se lee es un dígito, la categoría léxica va a ser **Número**, ya que un identificador no puede comenzar por un dígito. La MDD lee en este caso, comprobando según la gramática especificada anteriormente si el número es Entero o Real.

2.3. Comprobación de la gramática

A continuación explicaremos cómo la gramática con la que se ha trabajado es $LL(1)$. Tal y como se estudió en las clases de teoría, una gramática $LL(1)$ es una gramática que cumple las siguientes condiciones para cualquier par de reglas $A \to \alpha$ y $A \to \beta$:

$$primeros(\alpha) \cap primeros(\beta) = \emptyset$$
$$\epsilon \in primeros(\alpha) \implies primeros(\alpha) \cap siguientes(A) = \emptyset$$

Informalmente, esto implica que, en la tabla de símbolos, cada celda tiene como mucho una parte derecha: esto es, para cada combinación de no terminal a analizar y terminal en la entrada, sólo se puede avanzar siguiendo una sola regla. Para poder calcular la tabla de símbolos, antes es necesario calcular los **primeros** de cada regla y los **siguientes** de cada no terminal:

No terminales	**Primeros**
$\langle Programa \rangle$	{PROGRAMA}
$\langle decl\_var \rangle$	{VAR}
$\langle decl\_v \rangle$	{id}
$\langle lista\_id \rangle$	{id}
$\langle resto\_listaid \rangle$	{,}
$\langle Tipo \rangle$	{ENTERO,REAL,BOOLEANO},{VECTOR}
$\langle Tipo\_std \rangle$	{ENTERO},{REAL},{BOOLEANO}
$\langle instrucciones \rangle$	{INICIO}
$\langle lista\_inst \rangle$	{INICIO,id,LEE,ESCRIBE,SI,MIENTRAS}
$\langle instruccin \rangle$	{INICIO},{id},{LEE, ESCRIBE},{SI},{MIENTRAS}
$\langle Inst\_simple \rangle$	{id}
$\langle resto\_instsimple \rangle$	{opasigna},{[}
$\langle variable \rangle$	{id}
$\langle resto\_var \rangle$	{[}
$\langle inst\_e/s \rangle$	{LEE},{ESCRIBE}
$\langle expresin \rangle$	{id,num,(,NO,CIERTO,FALSO,+,-}
$\langle expresin' \rangle$	{oprel}
$\langle expr\_simple \rangle$	{id,num,(,NO,CIERTO,FALSO},{+,-}
$\langle resto\_exsimple \rangle$	{opsuma},{O}
$\langle termino \rangle$	{id,num,(,NO,CIERTO,FALSO}
$\langle resto\_term \rangle$	{opmult},{Y}
$\langle factor \rangle$	{id},{num},{(},{NO},{CIERTO},{FALSO}
$\langle signo \rangle$	{+},{-}

No terminales	**Siguientes**
$\langle Programa \rangle$	{$}
$\langle decl\_var \rangle$	{INICIO}
$\langle decl\_v \rangle$	{INICIO}
$\langle lista\_id \rangle$	{:}
$\langle resto\_listaid \rangle$	{:}
$\langle Tipo \rangle$	{;}
$\langle Tipo\_std \rangle$	{;}
$\langle instrucciones \rangle$	{.}
$\langle lista\_inst \rangle$	{FIN}
$\langle instruccin \rangle$	{;,SINO}
$\langle Inst\_simple \rangle$	{;,SINO}
$\langle resto\_instsimple \rangle$	{;,SINO}
$\langle variable \rangle$	{opmult,Y,opsuma,O,],),oprel,ENTONCES,HACER,;,SINO}
$\langle resto\_var \rangle$	{opmult,Y,opsuma,O,],),oprel,ENTONCES,HACER,;,SINO}
$\langle inst\_e/s \rangle$	{;,SINO}
$\langle expresin \rangle$	{ENTONCES,HACER,;,SINO}
$\langle expresin' \rangle$	{ENTONCES,HACER,;,SINO}
$\langle expr\_simple \rangle$	{],),oprel,ENTONCES,HACER,;,SINO}
$\langle resto\_exsimple \rangle$	{],),oprel,ENTONCES,HACER,;,SINO}
$\langle termino \rangle$	{opsuma,O,],),oprel,ENTONCES,HACER,;,SINO}
$\langle resto\_term \rangle$	{opsuma,O,],),oprel,ENTONCES,HACER,;,SINO}
$\langle factor \rangle$	{opmult,Y,opsuma,O,],),oprel,ENTONCES,HACER,;,SINO}
$\langle signo \rangle$	{id,num,(,NO,CIERTO,FALSO}

Así pues, considerando que el código es una implementación manual de la tabla de símbolos, se puede concluir que la gramática a analizar es $LL(1)$ ya que, para cada par (no-terminal, terminal), tan sólo se sigue aplicando una regla, sin excepciones. No obstante, la regla:

$$\langle expresion \rangle \quad \rightarrow \quad \langle expr\_simple \rangle \ \textbf{oprel} \ \langle expr\_simple \rangle | \langle expr\_simple \rangle$$

presenta prefijos comunes, por lo que fue necesario dividirla para eliminar dicho problema:

$$\langle expresion \rangle \quad \rightarrow \quad \langle expr\_simple \rangle \langle expresion' \rangle$$

$$\langle expresion' \rangle \quad \rightarrow \quad \textbf{oprel} \ \langle expr\_simple \rangle \ | \lambda$$

Una vez hecho esto, la gramática resultante ya es $LL(1)$, tal y como se refleja en el código.

3. Detalles de implementación

A la hora de implementar el sistema, se nos proporcionó una serie de ficheros con el esqueleto del código que tendríamos que completar. Dicho esqueleto contenía parte del analizador léxico, además de una implementación anterior del Árbol de Sintaxis Abstracta que se realizó el año anterior, y que nos ha servido de orientación para realizar nuestra propia implementación. Los ficheros proporcionados son:

- **componentes.py:** Es en este fichero donde se encuentran todas las clases que representan las distintas categorías léxicas. Sin embargo, las clases venían vacías, y se debía añadir el código necesario para almacenar la información necesaria (como el valor, en el caso de un Número) en cada clase.
- **flujo.py:** Contiene una clase que nos permite leer el flujo de entrada de un programa. Ya venía completo, así que no había que modificar este fichero

- **Prueba1.eje:** Este fichero contiene un pequeño programa de prueba para que se pueda testear el código que se va generando.
- **analex.py:** El esqueleto del analizador léxico. Contenía el constructor de la clase, una función main para testear el programa que se le pasara por parámetro, y un ejemplo de como analizar la entrada.

A partir de estos ficheros, se ha generado un compilador capaz de crear un Árbol de Sintaxis Abstracta del código que se especifica en la descripción de la práctica.

3.1. Analizador léxico

Esta parte de nuestro compilador es la que implementa la MDD que se ha especificado anteriormente. Gracias al archivo **flujo.py**, contamos con una función `siguiente()` que nos devuelve el siguiente caracter del fichero y una función `devuelve()` que nos permite devolver el caracter leído al flujo, para volver a analizarlo (en caso de que hayamos entrado en un camino erróneo)

Hemos creado la función `Analiza()`, que nos devuelve el componente léxico que viene a continuación. Para ello, lee el primer caracter para detectar la categoría léxica a la que pertenece con un `if-else`, ya que Python no permite la instrucción `switch`. Si el primer caracter es un espacio, o una llave de apertura (que nos marca el comienzo de un comentario) omite dicho caracter y sigue leyendo caracteres hasta que encuentra algo distinto del espacio (Listing 1) o la llave de cierre (Listing 2)

Listing 1: Omisión de espacios

```
1  if ch==" ":
2   while ch == " ":
3     ch = self.flujo.siguiente()
4   self.flujo.devuelve(ch)
5   return self.Analiza()
```

Listing 2: Tratamiento de comentarios

```
1  elif ch == "{":
2   while ch != "}":
3     ch = self.flujo.siguiente()
4   return self.Analiza()
```

Cuando el caracter que lee la función es un caracter alfabético, sigue leyendo todos los caracteres alfanuméricos que le siguen hasta encontrar un

espacio. Para ello, se hace uso de las funciones predefinidas de Python, que detectan los dígitos, caracteres, etc. Al completar la palabra, comprueba si se encuentra en la tabla de palabras reservadas, y devuelve la categoría léxica apropiada. (Listing 3)

Listing 3: Detección de palabras

```
1   elif ch.isalpha():
2    word = ""
3    while ((ch).isalnum()):
4      word += ch
5      ch = self.flujo.siguiente()
6    self.flujo.devuelve(ch)
7    if word in self.PR:
8      return componentes.PR(word, self.nlinea)
9    else:
10     return componentes.Identif(word, self.nlinea)
```

Cuando el caracter que se lee es un dígito, quiere decir que la categoría va a ser un Número. La función lee y guarda todos los dígitos hasta que se encuentra un espacio o un punto. En el primer caso, la función almacena que se trata de un Entero y devuelve la categoría correspondiente. En el segundo caso, la función comprueba que existe como mínimo un dígito después del punto, para almacenar que se trata de un Real; o devuelve un error. (Listing 4)

Listing 4: Detección de números

```
elif ch.isdigit():
num = ""
num+=ch
ch=self.flujo.siguiente()
while(ch.isdigit()):
   num+=ch
   ch=self.flujo.siguiente()
if (ch != '.'):
   self.flujo.devuelve(ch)
   return componentes.Numero(num, self.nlinea, 'ENTERO')
else:
   newCh = self.flujo.siguiente()
   if not (newCh.isdigit()):
      self.flujo.devuelve(newCh)
      self.flujo.devuelve(ch)
      print "ERROR: NUMERO REAL MAL ESPECIFICADO" # tenemos un
          comentario no abierto
      return self.Analiza()
   num+=ch
   num+=newCh
   ch=self.flujo.siguiente()
   while((ch).isdigit()):
      num+=ch
      ch=self.flujo.siguiente()
   self.flujo.devuelve(ch)
   return componentes.Numero(num, self.nlinea, 'REAL')
```

Por último, los caracteres no visibles como los saltos de línea o retornos de carro se tratan con su correspondiente codificación numérica, al no tener otra herramienta para tratarlos (Listing 5). Cualquier otro caracter no nulo hace que la función devuelva un error, mientras que al encontrar el primer caracter nulo se considera que se ha llegado al fin de fichero y se devuelve el componente correspondiente (Listing 6).

Listing 5: Caracteres no visibles

```
elif (ch is not '' and ord(ch) == 13):
   self.nlinea += 1
   return self.Analiza()
elif (ch is not '' and ord(ch) == 10):
   return self.Analiza()
```

Listing 6: Errores y fin de fichero

```
1   elif len(ch) is not 0:
2     print "ERROR: CARACTER NO DEFINIDO EN LA ESPECIFICACION DEL
          LENGUAJE"
3     return self.Analiza()
4   else:
5     return componentes.EOF()
```

3.2. Analizador sintáctico

Para la construcción del analizador sintáctico, no disponíamos de un esqueleto de código, pero si de unas instrucciones básicas en el material de clase de teoría. A partir de éstas, pudimos deducir la estructura del analizador sintáctico. En primer lugar, se han creado dos funciones que nos ayudarán con las comprobaciones. La primera es una función `avanza()` que nos devuelve el siguiente componente en nuestro programa, extraido del analizador léxico explicado en la sección anterior. La segunda función, `comprueba()`, nos realiza la comprobación de que un componente pertenece a una determinada categoría léxica, o devuelve un error.

La aproximación que se ha tomado a la hora de programar el analizador sintáctico es la creación de una función por cada no terminal. En estas funciones, se van comprobando uno a uno los elementos que deberían aparecer, según la especificación sintáctica proporcionada en el material de la práctica.

El primer paso es comprobar que el primer elemento leído pertenece a los primeros de la regla, que ya se han sacado y explicado anteriormente. Así, podemos detectar a qué regla ha entrado nuestro programa. Gracias a la función `comprueba()` definida anteriormente, se van comprobando todos los terminales que deberían aparecer. En caso de que el elemento siguiente sea un no terminal, se llama a la función de dicho no terminal. Si la regla de producción es una regla vacía, no se hace nada en ese paso. Se puede observar como se ha hecho la implementación del no terminal $\langle resto\_term \rangle$ (Listings 7)

$$
\begin{aligned}
\langle resto\_term \rangle &\rightarrow \textbf{opmult } \langle factor \rangle \langle resto\_exsimple \rangle \\
\langle resto\_term \rangle &\rightarrow \textbf{O } \langle factor \rangle \langle resto\_exsimple \rangle \\
\langle resto\_term \rangle &\rightarrow \lambda
\end{aligned}
$$

Listing 7: analizaRestoTerm()

```
1       def analizaRestoTerm(self):
2          if self.componente.cat == "OpMult":
3             self.avanza()
4             self.analizaFactor()
5             self.analizaRestoTerm()
6          elif self.componente.cat == "PR" and self.componente.valor
               == "Y":
7             self.avanza()
8             self.analizaFactor()
9             self.analizaRestoTerm()
10         elif (self.componente.cat == "PR" and self.componente.valor
               in ["ENTONCES", "HACER", "SINO", "O"]) or
               self.componente.cat == "ParentCi" or
               self.componente.cat == "CorCi" or self.componente.cat
               == "OpRel" or self.componente.cat == "PtoComa" or
               self.componente.cat == "OpAdd":
11            pass
12         else:
13            print "Error: SE ESPERABA Resto de Termino en linea " +
                  str(self.lexico.nlinea)
14            while not ((self.componente.cat == "PR" and
                  self.componente.valor in ["O", "ENTONCES", "HACER",
                  "SINO"]) or self.componente.cat == "OpRel" or
                  self.componente.cat == "OpAdd" or
                  self.componente.cat == "CorCi" or
                  self.componente.cat == "ParentCi" or
                  self.componente.cat == "PtoComa"):
15               self.avanza()
16            return
```

Para el tratamiento de errores, se ha optado por un tratamiento en modo pánico. Esto quiere decir que si el componente que encontramos no es el componente esperado, avanzamos y tiramos todos los componentes que vienen a continuación hasta llegar a uno de los siguientes del no terminal que ha dado el problema. A partir de ahí, podemos continuar el análisis de una manera normal, habiendo informado del problema. Ésto se observa muy bien en el código anterior (Listings 7), ya que la condición `else` muestra un bucle que va avanzando en el analizador léxico sin hacer nada con los componentes mientras el elemento no pertenezca a los siguientes de $\langle resto\_term \rangle$. Además, en el momento que encuentre un error, el compilador avisa de la línea de error y del tipo de error encontrado, mostrándolo por pantalla.

3.3. Analizador semántico

Por último, para la construcción del analizador semántico, se ha aprovechado el código del analizador sintáctico, y se han intercalado las operaciones semánticas pertinentes. Aunque todo lo expuesto a continuación es parte del análisis semántico, dada la diferencia entre sus flujos de información, hemos considerado necesario dividir la explicación del análisis semántico en dos partes: **comprobaciones semánticas** y **tabla de símbolos**.

3.3.1. Tabla de símbolos

En primer lugar, nuestro código lee las declaraciones de variables al principio del programa, y con dicha información construye la tabla de símbolos. Dicha tabla se utiliza posteriormente por el resto del analizador semántico para rellenar la información de los nodos pertinentes. La tabla de símbolos ha sido implementada por medio de un diccionario en Python, de tal forma que pueda ser indexada por el nombre de cada variable, y así se pueda acceder a su información. Para su construcción, tanto las funciones `analizaDeclVar` como `analizaDeclV` recuperan la lista de variables declaradas en cada línea (Listings 8), y si están repetidas, devuelve un mensaje de error. Si no, añade la variable a la tabla de símbolos, y sigue con el análisis. Por su parte, la función `analizaListaId` recupera el identificador de la primera variable, y lo añade a la lista que devuelve `analizaRestoListaId` (Listings 9). Esta lista, a su vez, será una lista con los identificadores del resto de variables, o una lista vacía (para acabar con la recursión).

Listing 8: analizaDeclVar()

```
def analizaDeclVar(self):
    if self.componente.cat == "PR" and self.componente.valor ==
        "VAR":
        # ...

        # RESTRICCION SEMANTICA: No puede haber identificadores
            repetidos
        for identif in listaIDs:
            if (identif in self.tablaSim):
                print "Error: no puede haber identificadores
                    repetidos. ID repetido: " + str(identif)
            else:
                self.tablaSim[identif] = tipo

        self.analizaDeclV()
    # ...
```

Listing 9: analizaListaId() y analizaRestoListaId()

```
def analizaListaId(self):
    if self.componente.cat == "Identif":
        identif = self.componente.valor
        self.avanza()
        restoIds = self.analizaRestoListaId()
        restoIds.append(identif)
        return (restoIds)

    # ...

 def analizaRestoListaId(self):
    if self.componente.cat == "Coma":
        self.avanza()
        restoIDs = self.analizaListaId()
        return restoIDs
    elif self.componente.cat == "DosPtos":
        return []

   # ...
```

Una vez construida la tabla de símbolos, otros métodos hacen uso de la misma para comprobar que las variables están declaradas antes de usarlas (Listings 10), o para comprobar que son de un tipo concreto (Listings 11).

Listing 10: analizaInstSimple()

```
1        if self.componente.cat == "Identif":
2          # RESTRICCION SEMANTICA: definir variables antes de usarlas
3            if (self.componente.valor not in self.tablaSim):
4                print "Error: variable no definida: '" +
                     self.componente.valor + "' en linea " +
                     str(self.componente.linea)
5            accVar = AST.NodoAccesoVariable(self.componente.valor,
                 self.lexico.nlinea,
                 self.tablaSim[self.componente.valor])
6            self.avanza()
7
8            # ...
```

Listing 11: analizaInstES()

```
1        if self.componente.cat == "PR" and self.componente.valor ==
             "LEE":
2            self.avanza()
3            self.comprueba("ParentAp")
4            # RESTRICCION SEMANTICA: el argumento de LEE solo puede
                 ser entero o real
5            if (self.tablaSim[self.componente.valor] not in
                 ["ENTERO", "REAL"]):
6                print "Error: el tipo a leer solo puede ser entero o
                     real (instruccion LEE en linea " +
                     str(self.componente.linea) + ")"
7
8            nodoLee = AST.NodoLee(self.componente.valor,
                 self.lexico.nlinea)
9         # ...
```

3.3.2. Comprobaciones semánticas

Para llevar a cabo tanto las comprobaciones semánticas habituales como la construcción del AST, se ha aprovechado la estructura de un fichero de Python ya existente creado anteriormente por el profesor: `AST.py`, y se ha adaptado para las necesidades concretas de nuestro lenguaje, dando lugar al fichero `ASTree.py`. Este fichero recoge los distintos tipos de nodos que puede

haber en el AST, y por cada nodo recoge la línea de código asociada, y sus atributos e hijos pertinentes. Además, cada nodo tiene métodos tanto para realizar sus comprobaciones semánticas y las de sus hijos como para mostrar información suya en forma de texto. Un ejemplo de nodo de dicho fichero se puede ver en (Listings 12).

Básicamente, las comprobaciones semánticas que se realizan son las indicadas en la descripción de la práctica, en concreto:

- No se pueden definir dos variables con el mismo nombre.
- Las variables se deben definir antes de usarlas.
- No puede haber identificadores que coincidan con las palabras reservadas del lenguaje (esto ya lo impide el analizador léxico).
- Conversión implícita de enteros en reales. Esto se hace dentro de las clases NodoAsignación, NodoAritmética y NodoComparación.
- No hay conversión implícita con el tipo booleano (se controla igual que antes).
- NodoLee solo acepta leer enteros y reales. Esto se comprueba dentro de NodoLee.
- NodoEscribe solo acepta expresiones. Esto ya está restringido por la sintaxis.

Para implementar el AST, hemos elegido representarlo como una lista dentro del analizador sintáctico. La función de análisis `analizaListaInst()` es la encargada de ir insertando nuevos nodos en dicha lista, uno por cada instrucción que lee. Dicha lista es luego retornada a `analizaInstrucciones()`, la cual usa el valor de retorno para construir el AST. Los tipos de instrucciones (y, por ende, de nodos) que acepta la primera función son los indicados en la descripción de la práctica, es decir: asignaciones, condicionales, bucles, lectura, escritura y sentencias compuestas. Para este último, crea como hijo una lista de sentencias. Para asignaciones, condicionales, bucles y nodos de escritura, puede darse el caso de que necesite analizar una expresión. En ese caso, creará los nodos de expresión que sean necesarios, los cuales pueden ser: operación aritmética, comparación; valor entero, real o booleano; acceso a variable o acceso a vector.

Listing 12: NodoAritmetico(AST)

```
class NodoAritmetico(AST):
   def __init__(self, izq, dcha, linea, op):
      # Aquí se inicializan los atributos e hijos del nodo, y se
          llama a compsem()

   def compsem(self):
      # Aquí se hacen las comprobaciones semánticas propias y las de
          los hijos

   def arbol(self):
      # Aquí se devuelve información del nodo y sus hijos en forma
          de string
```

4. Conclusiones

A lo largo de esta práctica se ha trabajado con Python para resolver los problemas típicos que se pueden encontrar a la hora de implementar un compilador, lo que nos ha permitido evitar los problemas típicos del desconocimiento de un lenguaje. Si bien otras opciones como puede ser Haskell resultan más apropiadas al generar un compilador real, por su eficiencia y otros factores, la dificultad de comprensión de algunos elementos necesarios como las mónadas hacen que la manera en que se ha desarrollado la práctica sea la más conveniente dado el temario de la asignatura.

Se ha podido implementar un compilador partiendo con apenas algunas herramientas que más que eliminar dificultad aceleran el proceso de construcción, ya que la herramienta `flujo.py` no tiene una gran dificultad pero hubiera requerido un tiempo que ha sido valioso para otras implementaciones.

Desde la gramática y especificaciones del lenguaje, se ha conseguido generar todos los pasos necesarios hasta crear el Árbol de Sintaxis Abstracta. La generación de código es un paso que, si bien no se ha tenido en cuenta en esta práctica, se puede realizar sin mayor problema una vez llegados a este punto. Por ello, la práctica ha resultado lo suficientemente completa, al permitirnos ver los problemas reales de los distintos análisis de un programa; sin llegar a ser extremadamente larga o compleja y no permitirnos comprender los conceptos de una manera adecuada.