Universidad de Castilla-La Mancha

Escuela Superior de Ingeniería Informática

Memoria de Prácticas:

Desarrollo de un Compilador Sencillo

Grado en Ingeniería Informática

Procesadores de Lenguajes

Profesor

Juan José Pardo Mateo

Alumnos

Domingo José Caballero Navarro

Rubén Castillo Carrasco

Contenido

INTRODUCCIÓN

A lo largo de esta memoria de prácticas, pondremos a prueba todos los conocimientos aprendidos a lo largo de la asignatura, implementando un compilador sencillo para un lenguaje de programación desde cero. Para ello, podemos dividir nuestra implementación en tres etapas distintas, la elaboración del analizador sintáctico, la elaboración del sintáctico y por último la del semántico, que nos permitirá obtener el AST del programa pasado en el fichero por parámetro a compilar.

ANALIZADOR LÉXICO

1. INTRODUCCIÓN Y CATEGORÍAS LÉXICAS A OBTENER

La implementación de este, que se corresponde con la primera etapa de la fase de análisis, tiene como objetivo separar la entrada de texto en distintos componentes léxicos, que serán usados en fases posteriores, además de la detección de errores en estos. Para nuestro compilador, hemos determinado las siguientes categorías léxicas:

- Palabras clave: formado por el conjunto {PROGRAMA, VAR, ENTERO, REAL, BOOLEANO, INICIO, FIN, SI, ENTONCES, SINO, MIENTRAS, HACER, LEE, ESCRIBE, Y, O, NO, CIERTO y FALSO}, escritas siempre en mayúscula.
- Identificador: formados por aquellas cadenas de letras y números, que comienzan siempre por letra. Cabe desatacar que, de este conjunto, hay que excluir al de las palabras clave.
- Número: que pueden ser tanto enteros como reales.
- Operador asignación: que se corresponde con la cadena de texto ‘:=’
- Operador suma: que se corresponde con los caracteres del conjunto {+,-}
- Operador multiplicación: que se corresponde con los caracteres del conjunto {*,/,%}
- Operador relacional: que se corresponde con los caracteres del conjunto { <, >, <>, <=, >=, =}
- Símbolos: que se conforma por los caracteres del conjunto {‘(’, ’)’, ’;’, ’.’, ’,’,’:’}
- Comentarios: aquellas cadenas que empiezan por ##, seguidas por cualquier tipo de carácter salvo el salto de línea y terminan con este último(\n). Como se verá posteriormente, estas al obtenerlas no tendrán uso alguno, sino que simplemente se desecharán.

Cabe destacar que otros caracteres, como \t, \n, \r o el espacio no compondrán categorías léxicas de por sí, sino que estas simplemente se usarán para separar componentes léxicas.

2. AUTÓMATA QUE CAPTURA NUESTRAS CATEGORÍAS LÉXICAS

A partir de estas podemos elaborar la MDD que captura los patrones de las categorías léxicas anteriores. Dado que es de un tamaño grande, de cara a explicar de forma detallada como captura el patrón de cada componente léxico, primero mostraremos de forma aislada el autómata que se basa en el patrón de cada una de las categorías.

Primero, el autómata que capturaría la categoría de números enteros sería el siguiente:

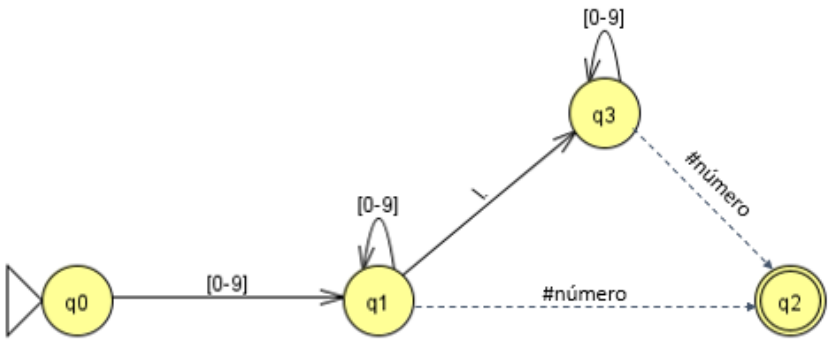

Imagen 1: autómata para la categoría léxica de números

Nota: la definición de la categoría léxica de número ha sido implementada acorde con la definición propuesta en el enunciado de las prácticas por el profesor, que facilita la implementación. Cabe destacar que esta captura ceros a la izquierda en enteros, como con 02, además de ceros a la derecha en números reales, como con 0.20. Este error, ya que nos basamos en la definición del profesor, no es tratado en nuestra implementación, pero sí que se almacena su valor correcto una vez capturado el componente léxico de tal forma que se almacena el valor de forma correcta en un atributo denominado valor. Además, hay que destacar que la categoría léxica es Número, en vez de tener una para enteros y otra para reales, almacenando este tipo en otra variable llamada tipo, como se verá posteriormente, y que evitará que tengamos que duplicar algunas reglas en el analizador sintáctico.

El autómata que capturaría la categoría léxica de palabras reservadas sería el siguiente:

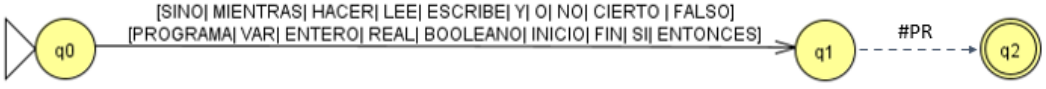

Imagen 2: autómata para la categoría léxica de palabra reservada

En cuanto a los identificadores, el autómata equivalente que los capturaría sería tal que:

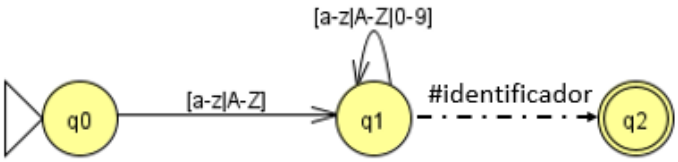

Imagen 3: autómata para la categoría léxica de identificador

Un autómata que capture operadores y símbolos para nuestro lenguaje sería de la forma:

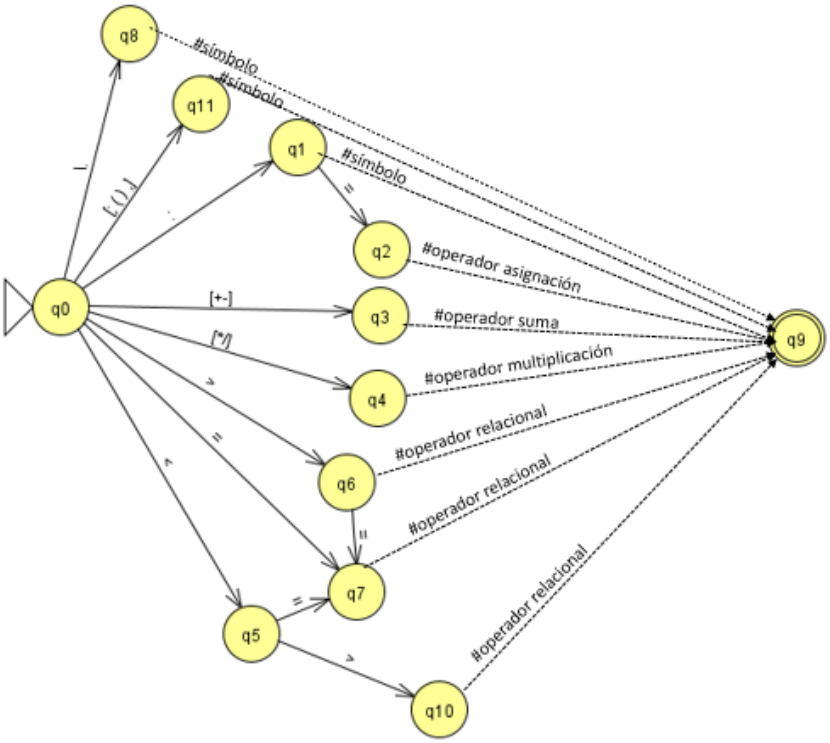

Imagen 4: autómata para las categorías léxicas de símbolo y operador relacional, de suma, multiplicación o asignación

Por último, un autómata que captura comentarios es de la forma:

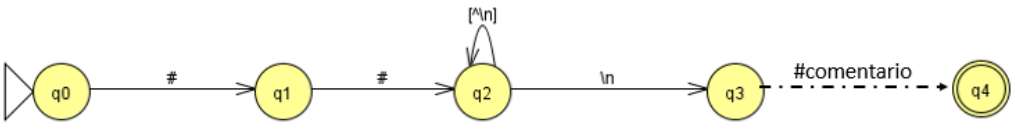

Imagen 5: autómata para la categoría léxica de comentario

3. IMPLEMENTACIÓN DE NUESTRO CÓDIGO

Ahora, a partir de la MDD definida anteriormente, hemos de implementarla para obtener nuestro analizador léxico.

Para cada una de las categorías léxicas definidas anteriormente en la MDD hemos de implementar en una clase independiente, de tal forma que también podamos almacenar el valor de los atributos necesarios para cada una de ellas. Es por ello por lo que tenemos una clase Componente, de la cual heredan las distintas categorías léxicas, y que almacenará la línea de cada componente léxica encontrada. Además, contiene un método para imprimir por pantalla los valores de los atributos de cada componente léxica. De esta heredan las siguientes clases, encargadas de almacenar las componentes léxicas que hemos considerado oportunas en nuestra implementación:

- Numero: donde se almacenan tanto los números reales como enteros. Contiene un atributo valor, que almacena el propio valor del número, es decir, su lexema, además de un atributo tipo, donde se almacena una cadena, cuyo valor puede ser o "Real" o "Entero". La implementación que seguimos sigue una estrategia "ambiciosa", donde cuando se busca una nueva componente léxica, si se detecta un número como primer carácter del lexema, llama al método TrataNum(), que busca el lexema en su totalidad de acuerdo con el autómata representado anteriormente. Nuestra implementación de este método es la siguiente:

```
def TrataNum(self,flujo, ch):
    l=ch
    real = False
    ch = self.flujo.siguiente()
    #Mientras encuentre un numero o bien el primer punto, sigue iterando
    while((ord(ch)<58 and ord(ch)>47) or (ch=='.' and real==False)):
        if(ch=='.'):
            #Encuentra punto, marcamos a True la variable para que, si vuelve a enocntrar otro pare el bucle,
            #al no considerarse este ultimo como parte del numero. Además nos sirve para crear la clase
            real=True
        l=l+ch
        ch=self.flujo.siguiente()
    #Cuando encontramos un caracter que no es del numero lo devolvemos al flujo para que se trate posteriormente
    flujo.devuelve(ch)
    if(real):
        return float(l)
    else:
        return int(l)
```

Imagen 6: Método trataNum()

Como se puede observar en el fragmento de código, cuando encuentra un carácter distinto de un número o más de un punto, devuelve el último carácter al flujo y pasa a buscar más componentes léxicas. Para identificar el tipo del carácter que nos encontramos, usamos el método ord(char), que, al pasarle un carácter, nos devuelve el valor de su código ASCII. Recordamos las correspondencias entre un carácter y su código mediante la siguiente tabla:

1	☺	17	◄	33	!	49	1	65	A	81	Q	97	a	113	q
2	☻	18	↕	34	"	50	2	66	B	82	R	98	b	114	r
3	♥	19	‼	35	#	51	3	67	C	83	S	99	c	115	s
4	♦	20	¶	36	$	52	4	68	D	84	T	100	d	116	t
5	♣	21	§	37	%	53	5	69	E	85	U	101	e	117	u
6	♠	22	▬	38	&	54	6	70	F	86	V	102	f	118	v
7	•	23	↨	39	'	55	7	71	G	87	W	103	g	119	w
8	◘	24	↑	40	(	56	8	72	H	88	X	104	h	120	x
9	○	25	↓	41	)	57	9	73	I	89	Y	105	i	121	y
10	◙	26	→	42	*	58	:	74	J	90	Z	106	j	122	z
11	♂	27	←	43	+	59	;	75	K	91	[	107	k	123	{
12	♀	28	∟	44	,	60	<	76	L	92	\	108	l	124	|
13	♪	29	↔	45	-	61	=	77	M	93	]	109	m	125	}
14	♫	30	▲	46	.	62	>	78	N	94	^	110	n	126	~
15	☼	31	▼	47	/	63	?	79	O	95	_	111	o	127	⌂
16	►	32		48	0	64	@	80	P	96	`	112	p	128	€

Imagen 7: tabla código ASCII

- Identif: sobre el cual hemos considerado necesario almacenar, además de su número de línea como en el resto de las categorías léxicas, su valor, que se corresponde con su lexema, al igual que sucede con la categoría léxica número. De nuevo, usamos un método auxiliar, que aparece en la imagen inferior, y donde se puede observar que, de nuevo, hacemos uso del método ord visto anteriormente.

```
def TrataIdent(self,flujo, ch):
    l = ch
    #Mientras que encuentre una letra, mayuscula o minuscula, o bien un numero sigue iterando
    while((ord(ch)<58 and ord(ch)>47) or (ord(ch)<90 and ord(ch)>64) or (ord(ch)<123 and ord(ch)>96)):

        ch=flujo.siguiente()
        if((ord(ch)<58 and ord(ch)>47) or (ord(ch)<90 and ord(ch)>64) or (ord(ch)<123 and ord(ch)>96)):
            l=l+ch
    #Devuelve el ultimo caracter, que no es una letra, para que sea tratado
    flujo.devuelve(ch)
    return l
```

Imagen 8: método trataIdentif()

- PR: donde almacenamos los mismos atributos que en Identif, y donde, en el atributo valor, almacenamos el lexema capturado, que equivale con un elemento del conjunto de palabras reservadas ya definido anteriormente. Esta, a excepción del resto de categorías léxicas, es capturada cuando hemos terminado de capturar el lexema de un identificador, y si este último coincide con un elemento del conjunto de palabras reservadas, se emite como PR en lugar de como Identif. Esto lo podemos ver en el siguiente fragmento de código:

```
elif(ch.isalpha()):
    cad = self.TrataIdent(self.flujo,ch)
    #Si una vez analizado el componente coincide con el nombre completo de una palabra reservada,
    #la guardamos como a estas
    if(cad in self.PR):
        o=PR(cad,self.nlinea)
    else:
        o=Identif(cad,self.nlinea)
    return o
```

Imagen 9: fragmento de código que comprueba que un identificador no es una PR

- OpRel: sobre el que se almacena el número de línea y otro atributo denominado valor, que se corresponde con el lexema. De nuevo, al igual que con el resto de las categorías léxicas, se captura siguiendo el autómata definido anteriormente.

- OpSum: categoría léxica para cuando encontramos un signo de suma o de resta, y sobre el cual almacenamos dos atributos, el número de línea, y el valor, que se corresponde con el carácter capturado.

- OpMult: equivalente al de OpSum pero para los caracteres de multiplicación, división y módulo.

- OpAsig: para el que solo será necesario almacenar el número de línea donde ha sido encontrado.

- Símbolo: sobre el que debemos almacenar el número de línea y, en su atribulo valor, el lexema.

- EOF: instanciada cuando encontramos el final del fichero y, por lo tanto, no hemos de almacenar ningún atributo sobre esta.

4. FUNCIONAMIENTO DEL ANALIZADOR LÉXICO

En nuestra implementación, este trata de obtener los componentes léxicos de forma voraz, buscando obtener los componentes con el lexema más largo posible, tomando caracteres de uno en uno del flujo hasta encontrar uno que no tenga una producción desde el estado del MDD en el que se encuentra.

La obtención de una componente léxica se lleva a cabo mediante el método Analiza(), al que llamamos desde el main. Este, cuando es llamado, toma el primer carácter del flujo y busca si cumple alguna producción de la MDD descrita anteriormente. Además, si este carácter es un espacio, \r, \n o \t, lo elimina y continua con el análisis del flujo. Si el carácter no puede conformar un componente léxico como el definido en la MDD, como lo sería el carácter "@", (se verá en profundidad en el próximo apartado) incrementa el contador de errores, indica por pantalla que hay un carácter invalido y prosigue con el análisis. Tras esto, en el propio bucle o en métodos como TrataIdent() o TrataNumero(), prosigue extrayendo caracteres siguiendo el esquema del MDD hasta que obtiene uno que no tiene producción, que devuelve al flujo para tratar de formar la siguiente componente léxica. El proceso llega a su fin cuando el flujo no contiene ningún carácter más, que hace que el método devuelva un componente de tipo EOF, que le indica al main que se ha llegado al fin del fichero.

Cabe destacar que se producen llamadas a métodos auxiliares, entre los que se encuentran:

- TrataComent: que, siguiendo las producciones de la MDD, cuando detecta un comentario va extrayendo caracteres hasta alcanzar el fin de línea, donde retorna para continuar con el análisis
- EliminaBlancos: tiene como objetivo eliminar todos los caracteres de espacio contiguos, finalizando cuando encuentra uno distinto a estos últimos.
- TrataNumero y TrataIdent: devuelven la instancia del componente léxico, haciendo más fácil la lectura del código al implementarse como funciones en lugar de estar en el propio bucle de la función Analiza()

5. TRATAMIENTO DE ERRORES

En nuestro código, serán considerados errores aquellos caracteres que no encuentran ninguna producción desde la MDD definida anteriormente (salvo los blancos y caracteres como \n, \r y \t, que sí que son tratados, pero no se emiten como componente léxico), como lo podría serlo el carácter '@'.

Para el tratamiento de estos, disponemos de un contador de errores, inicializado a 0, que, con cada carácter encontrado erróneo se incrementa, además de mostrar un mensaje por pantalla donde se indica que se ha encontrado un error.

En lo relativo a cadenas como “10.7.2”, aunque a priori podríamos pensar que se trata de un número real erróneo, al haber dos caracteres de punto en lo que a priori parece un único número, en esta fase del análisis no se detectan ningún error, sino que el análisis léxico nos devolvería tres componentes léxicas (un número con valor 10.7, un símbolo con valor ‘.’ Y un número con valor 2), y el error en este caso sería obtenido en otras fases de la compilación (en este caso, durante el análisis sintáctico). Lo mismo sucede con otras cadenas del flujo como “123a”, que en esta fase no devuelven error, a pesar de no existir un espacio que separe estos componentes léxicos, sino que nos devuelve dos componentes léxicas (número e identificador), que nos devolverán un error en fases posteriores.

Por último, cabe destacar que, cuando se llega al fin del flujo del archivo de entrada, se imprime el número total de errores por pantalla.

6. PRUEBAS DE NUESTRA IMPLEMENTACIÓN

Para realizar un testeo exhaustivo de nuestra implementación, hemos elaborado diversos archivos .txt que se usarán observar si se capturan la totalidad de categorías léxicas de forma correcta y para saber si se detecta y se responde de la forma deseada a los errores del fichero de entrada. Para ello, haremos uso de dos ficheros de texto, como se verá a continuación. Los dos archivos probados son los siguientes:

- Primer fichero de texto: proporcionado en el enunciado de la práctica. En este, aparecen una gran cantidad de categorías léxicas a detectar, pero no se introducen errores, de tal forma que se pueda comprobar que el analizador captura todas las componentes léxicas de forma correcta. El fichero en cuestión es de la forma:

```
PROGRAMA p1;
VAR i,x:ENTERO;
INICIO
      i:=0.0;
      MIENTRAS (i<>5) HACER
            INICIO
            x:=i*4;
            ESCRIBE(x);
      FIN;
FIN.
```

Imagen 10: fichero de prueba entrada1_analex.txt

Como se puede observar en la imagen inferior, la salida por línea de comandos al ejecutar nuestro código es la esperada, como podemos ver en la siguiente imagen:

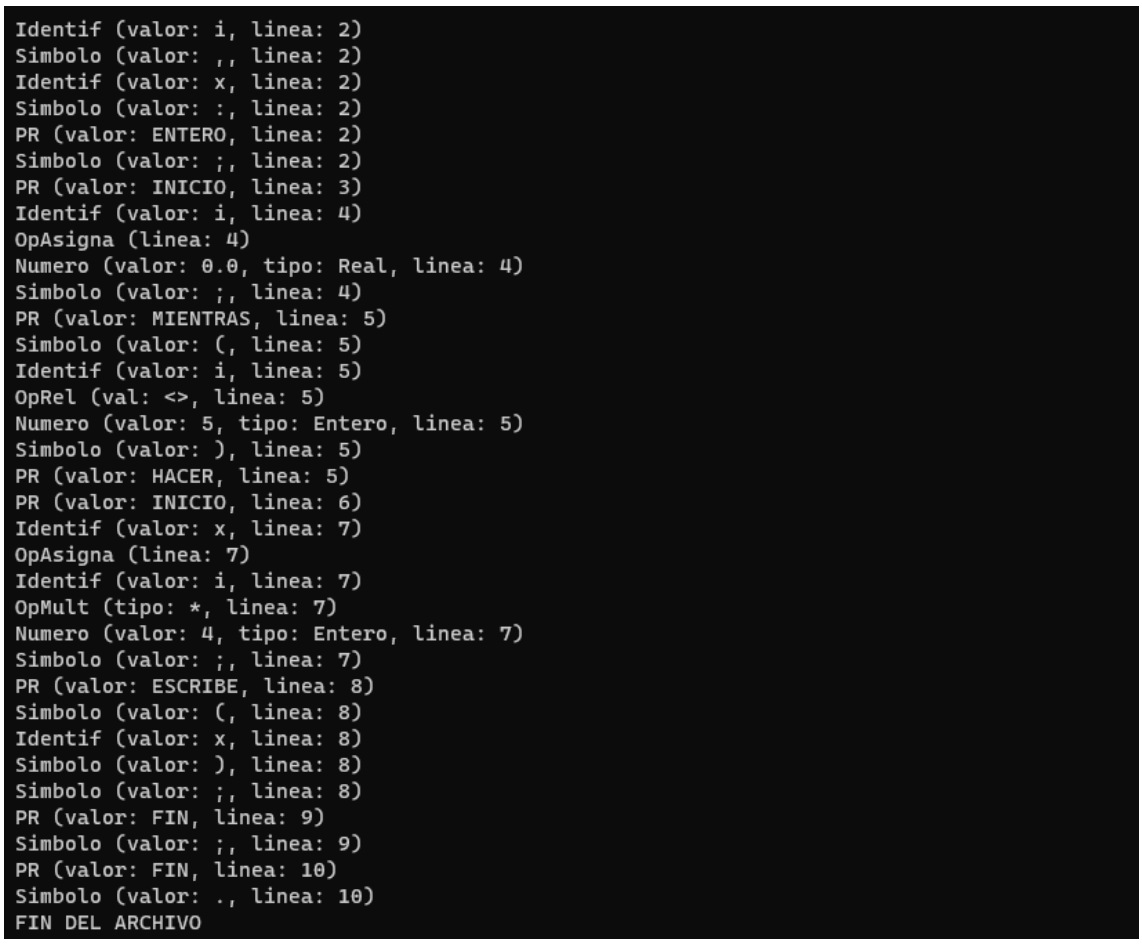

Imagen 11: salida del fichero de prueba entrada1_analex.txt

- Segundo fichero de texto: igual al anterior, donde se incluyen componentes léxicos adicionales para ser probados, además de errores como las cadenas de texto “p1@” o “#”, donde este último no tiene otra “#” tras él. También se incluye la cadena de texto “3.0.0” en la línea 4, que tal como hemos mencionado anteriormente, no devolverá error en esta fase del análisis, devolviendo un número un punto y otro número, pero si lo hará en fases posteriores como la del análisis sintáctico, ya que no podrá ir seguido un número de un símbolo de valor ‘.’. Por último, hemos añadido un lexema “INICIOoooo”, que, tal como hemos diseñado el analizador y su carácter ambicioso, nos devolverá una palabra reservada en lugar de separar el identificador de la palabra reservada. El fichero en cuestión, que se proporciona en nuestra práctica con el nombre de entrada2_analex, será por lo tanto de la siguiente forma:

```
PROGRAMA p1@;
VAR i,x:INTEGER¬;
INICIO
      i:=3.0.0;
      MIENTRAS (i<>5) HACER
            INICIO
            INICIOooooooo
                  x:=i*4;
                  ESCRIBE(x);
            FIN;
FIN.
```

Imagen 12: fichero de prueba entrada2_analex.txt

La salida, de nuevo, es la esperada, donde podemos detectar los errores de caracteres inválidos en la siguiente imagen:

```
ERROR LEXICO  Linea 1 ::  Caracter @ invalido
```

```
ERROR LEXICO  Linea 2 ::  Caracter # invalido
```

Imágenes 13 y 14 : fallos 1 y 2 en la salida del fichero de prueba entrada2_analex.txt

En cuanto al lexema “3.0.0”, el resultado también es el esperado:

```
Numero (valor: 3.0, tipo: Real, linea: 4)
Simbolo (valor: ., linea: 4)
Numero (valor: 0, tipo: Entero, linea: 4)
```

Imagen 15: fallo 3 en la salida del fichero de prueba entrada2_analex.txt

Y en lo relativo a la cadena “INICIOooooo”, también se producen los resultados esperados:

```
Identif (valor: INICIOooooooo, linea: 7)
```

Imagen 16: comportamiento en la salida del fichero de prueba entrada2_analex.txt

Por último, cabe destacar que, como era de esperar, encuentra un total de 2 errores

```
FIN DEL ARCHIVO
La cantidad de errores encontrados es de  2
```

Imagen 17: contador de errores en la salida del fichero de prueba entrada2_analex.txt

ANALIZADOR SINTÁCTICO

1. INTRODUCCIÓN

En esta fase, implementaremos un analizador sintáctico descendente, que, a partir de los componentes léxicos obtenidos por el analizador léxico, visto en la primera sección, nos permite encontrar las estructuras presentes en esta.

Estas estructuras se pueden representar mediante el árbol de análisis sintáctico, que explica cómo se puede derivar la cadena de entrada en la gramática que especifica el lenguaje.

2. GRAMÁTICA LL(1)

Para ello, nos basamos en la gramática dada en el enunciado de la práctica, que se muestra en la imagen inferior:

⟨Programa⟩ → **PROGRAMA id ;** ⟨decl_var⟩ ⟨instrucciones⟩ **.**
⟨decl_var⟩ → **VAR** ⟨lista_id⟩ **:** ⟨tipo_std⟩ **;** ⟨decl_v⟩ |λ
⟨decl_v⟩ → ⟨lista_id⟩ **:** ⟨tipo_std⟩ **;** ⟨decl_v⟩ |λ
⟨lista_id⟩ → **id** ⟨resto_listaid⟩
⟨resto_listaid⟩ → **,** ⟨lista_id⟩ |λ
⟨Tipo_std⟩ → **ENTERO** | **REAL** | **BOOLEANO**
⟨instrucciones⟩ → **INICIO** ⟨lista_inst⟩ **FIN**
⟨lista_inst⟩ → ⟨instrucción⟩ **;** ⟨lista_inst⟩ |λ
⟨instrucción⟩ → **INICIO** ⟨lista_inst⟩**FIN** | ⟨inst_simple⟩ | ⟨inst_e/s⟩
→ **SI** ⟨expresion⟩ **ENTONCES** ⟨instruccion⟩ **SINO** ⟨instrucción⟩
→ **MIENTRAS** ⟨expresión⟩ **HACER** ⟨instruccion⟩
⟨Inst_simple⟩ → **id opasigna** ⟨expresión⟩
⟨inst_e/s⟩ → **LEE (id)** | **ESCRIBE (** ⟨expr_simple⟩**)**
⟨expresión⟩ → ⟨expr_simple⟩ **oprel** ⟨expr_simple⟩ | ⟨expr_simple⟩
⟨expr_simple⟩ → ⟨término⟩ ⟨resto_exsimple⟩ | ⟨signo⟩ ⟨término⟩ ⟨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⟩ → ⟨id⟩ | **num** | **(** ⟨expresión⟩ **)** | **NO** ⟨factor⟩ | **CIERTO** | **FALSO**
⟨signo⟩ → +|−

Imagen 18: gramática proporcionada en el enunciado de la práctica

Es observable que, en la mayoría de las reglas, existen disyunciones, algo que debemos de eliminar. Otro añadido que hace que esta no esté en LL(1) es las producciones del no terminal <Expresión>, donde existen prefijos comunes. Este fallo quedaría eliminado sustituyéndolo por las siguientes reglas equivalentes:

```
<expresion> → <expr_simple> <expresion_aux>
<expresion_aux> → oprel <expr_simple>
<expresion_aux> → λ
```

Imagen 19: creación del NT <expresión_aux> para convertirla a LL1

Una vez realizado lo anterior, la **gramática resultante** sí que se encuentra en LL(1), y esta la hemos **almacenado** para que sea más fácil de visualizar en un fichero de texto llamado **primeros_siguientes.txt**, donde aparece junto con los primeros de la gramática del siguiente apartado.

3. TABLA DE PRIMEROS Y SIGUIENTES

Una vez realizado esto, para llevar a cabo nuestra implementación, es necesario obtener la tabla de primeros y los siguientes de aquellos no terminales anulables. En nuestro caso, hemos hallado los primeros y los siguientes de todos los no terminales. Esta la hemos almacenado en el fichero de texto anterior **primeros_siguientes.txt.**

Una vez elaborada esta, para ayudarnos en nuestra implementación hemos elaborado una **tabla de análisis**. Esta, a diferencia de la de primeros y siguientes, la hemos elaborado en un archivo .xlsx para que sea más fácil de seguir. Esta la podemos encontrar con el nombre **tabla_analisis.xlsx**

4. DETALLES DE LA IMPLEMENTACIÓN

Para facilitar la implementación, minimizar la cantidad de errores y facilitar la corrección de estos, hemos dividido nuestro código en diversas secciones, las cuales explicaremos a continuación de forma individual:

Función main:

```
if __name__=="__main__":
    script, filename=argv
    txt=open(filename)
    print ("Este es tu fichero %r" % filename)
    i=0
    fl = flujo.Flujo(txt)
    anlex=analex.Analex(fl)
    S = Sintactico(anlex)
    if S.Programa():
      print ("Analisis sintactico SATISFACTORIO. Fichero :", filename, "CORRECTO")
    else:
       print ("Analisis sintactico CON ERRORES. Fichero :", filename, "ERRONEO")
```

Imagen 20: función main de nuestro analizador sintáctico

Aquí se inicializa al analizador léxico, para que este nos vaya devolviendo las componentes léxicas presentes en el fichero de entrada cuando se lo solicitemos y para obtener el valor del primer siguiente. Una vez hecho esto, se llama a la función Programa(), que se corresponde con primera regla por la cual se comienza el análisis sintáctico.

Listado de errores:

Estos se lanzan cuando estamos en una regla y el siguiente no encaja con ninguno de los siguientes esperados. Dentro de estos, destacamos dos errores distintos, por un lado, los errores al llegar a un terminal, que tienen lugar cuando llegamos a un no terminal y el valor del siguiente no encaja con el valor se este, como por ejemplo los errores de la imagen inferior:

```
if nerr == 1:
    print ("Linea: " + str(self.token.linea) + "  ERROR Se espera PROGRAMA")
elif nerr==2:
    print ("Linea: " + str(self.token.linea) + "  ERROR:Se espera IDENTIFICADOR")
elif nerr==3:
    print ("Linea: " + str(self.token.linea) + "  ERROR:Se espera ;")
elif nerr==4:
    print ("Linea: " + str(self.token.linea) + "  ERROR:Se espera .")
elif nerr==5:
    print ("Linea: " + str(self.token.linea) + "  ERROR:Se espera :")
```

Imagen 21: errores de terminales

Por otro lado, tenemos los errores lanzados cuando llegamos a un no terminal y el valor del siguiente no está entre los siguientes esperados para ese no terminal. Ejemplos de estos los encontramos en la imagen inferior:

```
elif nerr==16:  #Se espera decl_var
    print ("Linea: " + str(self.token.linea) + "  ERROR:Se espera PR(VAR o INICIO)")
elif nerr==17:  #Se espera decl_v
    print ("Linea: " + str(self.token.linea) + "  ERROR:Se espera ID o INICIO")
elif nerr==18:  #Se espera lista_id
    print ("Linea: " + str(self.token.linea) + "  ERROR:Se espera ID")
elif nerr==19:  #Se espera resto_lista_id
    print ("Linea: " + str(self.token.linea) + "  ERROR:Se espera ; o .")
elif nerr==20:  #Se espera tipo_std
    print ("Linea: " + str(self.token.linea) + "  ERROR:Se espera PR(ENTERO REAL o BOOLEANO)")
elif nerr==21:  #Se espera instrucciones
    print ("Linea: " + str(self.token.linea) + "  ERROR:Se espera INICIO")
```

Imagen 22: errores de no terminales

SI observamos nuestra tabla de análisis, podemos ver que, cuando estamos en <tipo_std>, solo podemos lanzar reglas si nuestro siguiente es del conjunto {ENTERO, REAL, BOOLEANO}, y, por ello, la estructura del error 20.

Es por ello por lo que será necesario un error para cada uno de los terminales y de los no terminales de nuestro programa. (Aunque se podrían reciclar algunos errores en No terminales si el valor de los siguientes esperados coinciden).

Llegada a un terminal/ no terminal

Para lanzar un error si no encontramos un siguiente valido para alguna producción en la llegada a un no terminal o para, en el caso de los terminales, lanzar un erro si el siguiente no se corresponde con el esperado, hemos implementado una serie de funciones que, además, para el caso de los no terminales, nos enviaran a la regla correspondiente atendiendo al valor del siguiente. Es por ello por lo que hemos implementado una función de este tipo tanto para cada terminal como para cada no terminal

Unos ejemplos de estas funciones para los terminales son los de la siguiente imagen, que además de lanzar el error, una vez comprobado que el valor del siguiente coincide, se llama al analizador léxico para que este nos devuelva el siguiente componente.

```
def var(self):
    if(self.token.cat=="PR" and self.token.valor=="VAR"):
        self.token = self.lexico.Analiza()
    else:
        self.Error (34,self.token)
        return False
    return True

def id(self):
    if(self.token.cat=="Identif" ):
        self.token = self.lexico.Analiza()
    else:
        self.Error (2,self.token)
        return False
    return True
```

Imagen 23: funciones de llegada a un terminal

Para los no terminales, alguna implementación de este tipo de funciones se puede observar en la siguiente imagen:

```
def decl_var(self):
    if(self.token.cat=="PR" and self.token.valor=="VAR"):
        if  not(self.decl_var_2()): return False
    elif(self.token.cat=="PR" and self.token.valor=="INICIO"):
        if  not(self.decl_var_3()): return False
    else:
        self.Error (16,self.token)
        return False
    return True

def decl_v(self):
    if(self.token.cat=="Identif" ):
        if  not(self.decl_v_4()): return False
    elif(self.token.cat=="PR" and self.token.valor=="INICIO"):
        if  not(self.decl_v_5()): return False
    else:
        self.Error (17,self.token)
        return False
    return True
```

Imagen 24: funciones de llegada a un no terminal

Implementación de las reglas de la gramática

Para implementar cada una de las reglas de la gramática, hemos elaborado una función para cada una de ellas. Para distinguirlas de las anteriores, las hemos llamado de la forma noTerminal_numeroRegla(). Desde estas, siguiendo la estructura de las reglas de nuestro fichero primeros_siguientes.txt, se van haciendo llamadas a las funciones descritas anteriormente, de tal forma que se va comprobando que la estructura sintáctica del programa es la adecuada. El proceso llegará a su fin en una regla cuando se haya llegado al fin de estas sin errores. Cabe destacar que, cuando una regla es vacía, simplemente devuelve cierto. Un ejemplo de la implementación de este tipo de funciones lo vemos en la imagen inferior:

```
def decl_var_2(self):
    if not(self.var()): return False
    if not(self.lista_ids()): return False
    if not(self.dos_puntos()): return False
    if not(self.tipo_std()): return False
    if not(self.punto_coma()): return False
    if not(self.decl_v()): return False
    return True


def decl_var_3(self):
    return True
```

Imagen 25: funciones de las reglas de nuestra gramática

Gestión de errores

En esta fase del análisis, para no complicarlo de forma excesiva, hemos considerado que, cuando se alcance un error, es decir, no dado nuestro siguiente este no tenga el valor deseado, simplemente el programa se dejará de ejecutar, informando de que ha habido un error en este. Un programa por lo tanto será cierto si, una vez alcanzado el final de la primera regla, el valor de la categoría semántica del siguiente es EOF, que indica el fin del fichero

Nota: esto cambiará en la fase del análisis semántico, donde se tratará de avanzar lo máximo posible en el análisis, donde, a pesar de encontrar fallos no se finalizará la ejecución del análisis.

5. PRUEBAS DE NUESTRA IMPLEMENTACIÓN

Para ello, a diferencia de la fase anterior donde se hizo uso de dos ficheros, se hará uso de tres, para probar la mayor cantidad de errores posibles. Los resultados del testeo para estos han sido los siguientes:

- Primer fichero de prueba: fichero con el nombre entrada1_anasint.txt, cuyo contenido es igual al proporcionado en el enunciado de la práctica, como se puede observar en la imagen inferior:

```
PROGRAMA p1;
VAR i,x:ENTERO;
INICIO
      i:=0.0;
      MIENTRAS (i<>5) HACER
            INICIO
            x:=i*4;
            ESCRIBE(x);
      FIN;
FIN.
```

Imagen 26: fichero de prueba entrada1_anasint.txt

Para testear el correcto funcionamiento de este, hemos añadido comentarios en cada una de las reglas para saber en cual estamos. El resultado al ejecutarlo es el siguiente:

```
resto_expsimple_30
expresion_aux_25
intrsuccion_15
lista_inst_13
intrsuccion_16
isnt_simple_20
expresion23
expr_simple_26
termino31
factor_35
resto_term_32
factor_36
resto_term_34
resto_expsimple_30
expresion_aux_25
lista_inst_13
intrsuccion_17
ins_e_s_22
expr_simple_26
termino31
factor_35
resto_term_34
resto_expsimple_30
lista_intr_14
lista_intr_14
FIN DEL ARCHIVO
La cantidad de errores encontrados es de  0
Analisis sintactico SATISFACTORIO. Fichero : entrada1_anasint.txt CORRECTO
```

Imagen 26: salida fichero de prueba entrada1_anasint.txt

Podemos con esto podemos afirmar que, al menos para el fichero dado, el analizador se comporta de la forma esperada, encaminándose por las reglas adecuadas atendiendo al fichero de entrada.

- Segundo fichero de prueba: tomando el fichero anterior, introducimos un error en la instrucción mientras de la línea 5, eliminando la condición del bucle, como se puede ver en la imagen inferior:

```
PROGRAMA p1;
VAR i,x:ENTERO;
INICIO
      i:=0.0;
      MIENTRAS HACER
            INICIO
            x:=i*4;
            ESCRIBE(x);
      FIN;
FIN.
```

Imagen 27: fichero de prueba entrada2_anasint.txt

La salida al ejecutarlo es la siguiente:

```
(base) C:\Users\Rubén\Escritorio\analex>py anasint.py entrada2_anasint.txt
Este es tu fichero 'entrada2_anasint.txt'
programa
decl_var_2
lista_id_6
resto_lista_id_7
resto_lista_id_7  Identif   x
lista_id_6
resto_lista_id_8
tipo std 9
decl var 5
intrsucciones_12
lista_inst_13
intrsuccion_16
isnt_simple_20
expresion23
expr_simple_26
termino31
factor_36
resto_term_34
resto_expsimple_30
expresion_aux_25
lista_inst_13
intrsuccion_19
Linea: 5  ERROR:Se espera ID NO CIERTO FALSO  + - o (
Analisis sintactico CON ERRORES. Fichero : entrada2_anasint.txt ERRONEO
```

Imagen 28: salida del fichero de prueba entrada2_anasint.txt

Como podemos observar, el error lanzado se corresponde con el esperado, ya que este, como se puede ver en el código, es el correspondiente al del no terminal <expresión> y, por lo tanto, el comportamiento es adecuado.

- Tercer fichero de texto: por último, de cara a probar tanto el correcto funcionamiento de la omisión de la declaración de variables, como para probar el lanzamiento de un error en algún terminal, eliminando el identificador de alguna instrucción, hemos elaborado el siguiente fichero de texto a partir de nuestro primer fichero:

```
PROGRAMA p1;
INICIO
      i 0.0;
      MIENTRAS (i<>5) HACER
            INICIO
            x:=i*4;
            ESCRIBE(x);
      FIN;
FIN.
```

Imagen 29: fichero de prueba entrada3_anasint.txt

La salida para este es la siguiente:

```
(base) C:\Users\Rubén\Escritorio\analex>py anasint.py entrada3_anasint.txt
Este es tu fichero 'entrada3_anasint.txt'
programa
decl_var_3
intrsucciones_12
lista_inst_13
intrsuccion_16
isnt_simple_20
Linea: 3  ERROR:Se espera opasigna
Analisis sintactico CON ERRORES. Fichero : entrada3_anasint.txt ERRONEO
```

Imagen 30: fichero de prueba entrada3_anasint.txt

Con esto podemos observar que la salida es la esperada, ya que no salta ningún error al omitir la declaración de variables (de la que se encargará el analizador semántico), mientras que si aparece uno al detectar la falta del terminal opasig en la línea 3.

ANALIZADOR SEMÁNTICO

1. INTRODUCCIÓN

Por último, de cara a corregir errores no comprobados en fases anteriores, implementaremos un analizador semántico, que además de hacer esto último, generará el AST, que podrá ser usado en la fase de síntesis de nuestro lenguaje de programación. Esta fase, por lo tanto, podemos dividirla en distintos apartados:

2. RESTRICCIONES SEMÁNTICAS A COMPROBAR

1. No se podrán definir dos objetos distintos con el mismo nombre: esto lo solucionaremos haciendo uso de una tabla de símbolos, donde almacenamos todos los identificadores declarados y su lista de atributos. Cada vez que se declara un atributo, comprobamos que no existe uno con su nombre, y, si no lo hay, lo añadimos a la tabla de símbolos. En caso contrario, devolveremos un error. La implementación la veremos en profundidad en etapas posteriores.

2. No podrá haber identificadores que coincidan con palabras clave del lenguaje: esta no es necesaria implementarla en nuestro código ya que en el análisis léxico, si un lexema coincide con el de una palabra reservada se emite una palabra reservada en lugar de un identificador.

3. Se producirá una conversión implícita de números enteros en reales, tanto en expresiones como en asignaciones. Esta, en el caso de las expresiones, devolviendo en el tipo de nodo de una operación aritmética el tipo real. En cuanto a las asignaciones, el nodo será de tipo real a no ser que la variable a asignar sea de tipo booleano y el argumento un valor de tipo booleano.

4. La representación interna de los valores booleanos será 0 para FALSO y cualquier otro número para CIERTO. Esta representación se usará en operaciones entre números y booleanos, como las comparaciones, donde al booleano le tendremos que otorgar un valor numérico para que la operación pueda ser llevaba a cabo.

5. No habrá conversión implícita de tipos con el tipo booleano. Esto lo solucionaremos en el análisis sintáctico, haciendo uso de un atributo llamado "tipo" para los NT de tal forma que, si aplicamos operaciones, solo los booleanos pueden usar las propias (Y, O, NO). En el caso de que una expresión con tipo numérico trate de hacer uso de este tipo de operaciones, en lugar de hacer uso de un nodo OpBooleano, devolvemos un NodoVacio. Solo obtendremos un booleano de una expresión cuando el operador sea booleano y sus argumentos también. En el caso de usar un operador booleano y argumentos numéricos siempre se nos devolverá un error. En cambio, de una expresión que use un operador relacional, siempre el tipo devuelto será booleano.

6. La semántica de la instrucción LEE implica que el argumento solo puede ser una variable de tipo simple (ENTERO o REAL), mientras que la instrucción ESCRIBE permite como argumentos las expresiones ("expresión"). Lo comprobaremos en la fase del análisis sintáctico, apoyándonos en la tabla de símbolos, donde comprobamos que en lee() la variable ha sido declarada y tiene como tipo entero o real. En escribe dejaremos libertad y su resultado será cualquier elemento posible obtenible mediante el NT expresión.

3. OPERACIONES Y TIPOS

Por último, hemos considerado oportuno explicar los tipos de datos frente a los distintos operadores de nuestro lenguaje atendiendo a las restricciones anteriormente definidas. Para los tipos de operaciones oprel, los argumentos podrán ser de cualquier tipo, numéricos o booleanos, donde si alguno de ellos es booleano se hará uso de su representación interna numérica, además, esta operación nos devolverá siempre un tipo booleano.

En cuando a los tipos de operaciones opsum y opmult, los argumentos podrán ser de cualquier tipo, y, de nuevo, en el caso de los booleanos, se usará su representación numérica, y, con este tipo de operaciones, a diferencia de con oprel, el resultado será numérico.

Por último, para los operadores Y, O y NO, los argumentos siempre tendrán que ser booleanos, y como resultado nos devolverán un booleano.

4. NODOS USADOS EN NUESTRO AST

Para la generación del AST de nuestra gramática, hemos visto necesarios el uso de diversos tipos de nodos, donde nos hemos basado para ello en el conjunto de los nodos aportados en el enunciado de la práctica a los que le hemos añadido otros que considerábamos oportunos. Podemos dividir el conjunto de nodos en nodos para representar sentencias y nodos para representar expresiones. En los nodos para representar sentencias encontramos:

- Asignación: usada para la sentencia de asignación
- Si: usadas como sentencias de tipo condicional.
- Mientras: usadas como sentencias de tipo bucle.
- Escribe: donde como árbol hijo almacenará una expresión.
- Compuesta: sentencia usada para almacenar conjuntos de sentencias, donde, en sus hijos, almacenaremos una lista de nodos mayor a 1.
- Lee: donde como hijo almacenará el acceso a una variable, que hemos de comprobar que es de tipo numérica.

Entre los nodos para representar expresiones encontramos:

- Comparación: usado con los operadores relacionales.
- Aritmética: usada con las operaciones de tipo suma y multiplicación.
- BooleanaBinaria: usada para los operadores booleanos Y y O.
- BooleanaUnaria: usada para el operador booleano NO.
- Entero: usado por los números enteros, donde cabe destacar que en el atributo valor almacenamos su atributo en real.
- Real: que se comporta de la misma forma que el anterior.
- AccesoVariable: donde, antes de instanciarlo, hemos de comprobar que la variable ha sido declarada.

Además, se hace uso de un nodo vacío usado en el caso de encontrar un error.

5. ETDS EQUIVALENTE

Dadas las restricciones expuestas anteriormente y los tipos de nodos también definidos, podemos pasar a diseñar un esquema de traducción dirigido por la sintaxis que nos permita realizar el análisis semántico de nuestro lenguaje de la forma adecuada. El ETDS en cuestión lo hemos almacenado en un fichero llamado **etds.txt**

6. IMPLEMENTACIÓN DEL ANALIZADOR SEMÁNTICO

De cara a implementar el analizador semántico deberemos incorporar a nuestro programa dos tareas, por un lado, el tratamiento de la tabla de símbolos, y por otro los detalles de implementación adicionales:

6.1 TRATAMIENTO DE LA TABLA DE SIMBOLOS

Para realizar una serie de comprobaciones semánticas relativas al uso de variables es necesario el uso de una tabla de símbolos. En nuestro caso, todo lo relativo a ella lo llevaremos a cabo en nuestro analizador sintáctico. Para ello, haremos uso como estructura de datos de un diccionario en python, que, a partir del lexema del identificador nos devolverá el objeto almacenado, que en este caso hemos decidido que sea una lista de la forma [lexema, tipo, valor], y que almacenaremos en la regla de la declaración e iremos modificando su valor cuando este lo haga en reglas posteriores. Para el tratamiento de esta estructura de datos de una forma más sencilla, hemos implementado una serie de métodos:

- Inserta_tabla_simbolos(id, tipo): método que permite insertar una nueva lista en la tabla de símbolos al que se puede acceder mediante su lexema. A este se accederá en la fase de declaración de variables de la parte del análisis sintáctico. Además, cabe destacar que almacena los valores por defecto que hemos considerado oportunos para cada tipo de datos, siendo estos 0.0 para las variables numéricas y False para las de tipo Booleano. Podemos observar su implementación en la imagen inferior:

```
#Metodo que permite almacenar una nueva entrada en la tabla de simbolos
def inserta_tabla_simbolos(self,id,tipo):
    if not(self.esta_tabla_simbolos(id)):
        if(tipo=="Booleano"):
            self.tabla_simbolos[id] = [id,tipo,False]
        else:
            self.tabla_simbolos[id] = [id,tipo,0.0]
    else:
        return False
    return True
```

Imagen 31: función de inserción en la tabla de símbolos

- esta_tabla_simbolos(id): nos devuelve si un identificador está en la tabla de símbolos, y, por lo tanto, ha sido ya declarado o no. Se corresponde con el método inferior:

```
#Metodo que permite comprobar si hay un elemento con un id en la tabla de simbolos
def esta_tabla_simbolos(self,id):
    return (id in self.tabla_simbolos.keys())
```

Imagen 33: función de existencia en la tabla de símbolos

- modifica_valor_tabla(id,valor): nos permite modificar el valor de un identificador. Ha sido implementado, aunque en esta fase no se llega a utilizar. En cambio, en fases de síntesis sería útil al tratar con los nodos de asignación. Su código es el siguiente:

```
#Metodo que permite modificar el valor de una entrada de la tabla de simbolos
def modifica_valor_tabla(self,id,valor):
    if(self.esta_tabla_simbolos(id)):
        self.tabla_simbolos[id][2]=valor
        return True
    else:
        return False
```

Imagen 34: función de modificación del valor en una tabla de símbolos

Los dos primeros métodos son usados entre las reglas 2 y 11, donde debemos almacenar la lista de identificadores obtenidos mediante el no terminal <lista_id>, almacenándola con el tipo obtenido del no terminal <tipo_std>. En nuestro código, el almacenamiento de estas en la tabla tiene lugar al final de las funciones decl_var_2 y decl_v_4. La primera de estas la podemos observar en la siguiente imagen:

```
def decl_var_2(self):
    print("decl var 2 ", self.token.cat," ",self.token.valor)
    if not(self.var()): return False

    correcto,lista_ident=self.lista_ids([])
    if not(correcto): return False
    print("Lista indent\n",lista_ident)

    if not(self.dos_puntos()): return False
    print("decl var 2 ", self.token.cat," ",self.token.valor)

    correcto,tipo_std = self.tipo_std()
    if not(correcto):return False
    print("El tipo es ",tipo_std)

    for i in lista_ident:
        if(not(self.esta_tabla_simbolos(id))):
            if(self.inserta_tabla_simbolos(i,tipo_std)):
                print("Inserccion con exito")
            else:
                print("Fallo en la insercion")
        else:
            print("Fallo, este identificador ya existe, no puedes volver a declararlo")

    if not(self.punto_coma()): return False
    if not(self.decl_v()): return False
    return True
```

Imagen 35: ejemplos de uso de las funciones de la tabla de símbolos

Podemos observar que, en caso de fallo, no se detiene la ejecución del programa, sino que simplemente se imprime por pantalla información de que ha ocurrido un error en la inserción. Esto es debido a que buscamos, como se ha podido ver con la inclusión de los nodos vacíos, no el hecho de frenar el programa en cuanto encontremos un error, sino tratar de avanzar lo máximo posible.

Por último, cabe destacar que existen otras dos reglas que hacen uso de la tabla de símbolos en nuestra implementación, solo que estas lo hacen únicamente tras tener lugar la declaración de variables globales. Estas son la 21, que se corresponde con <inst_e/si> → LEE (id) y la 35, que se corresponde con <factor> → id.

Estas darán lugar a un nodo llamado AccesoVariable, y para instanciar este último, es necesario poder acceder a la lista almacenada en nuestro diccionario. Para acceder a esta, en nuestra implementación lo haremos primero comprobando que existe un identificador en la tabla con ese lexema, con el método esta_tabla_simbolos(id) y, una vez comprobado esto, con una instrucción de la forma self.tabla_simbolos[id], que nos devolverá la tupla que almacena el diccionario.

6.2 DETALLES DE LA IMPLEMENTACIÓN ADICIONALES

Una vez implementado el control de código correspondiente para la gestión de la declaración de variables, es necesario implementar el resto de código que permita, tanto detectar el resto de los errores semánticos definidos anteriormente, como instanciar los nodos necesarios que nos permitan obtener el AST de nuestra implementación. Todo esto lo hemos añadido en nuestro analizador sintáctico, donde, para poder ver todos los cambios entre este y el semántico, hemos creado una copia llamada anasem.py, donde tendrán lugar todos los cambios, de tal forma que conservemos el analizador sintáctico como tal.

En lo relativo al AST, este se iniciará con el árbol obtenido a partir del no terminal <instrucciones>, que contendrá o bien un NodoVacio() si no hay ninguna instrucción o bien el árbol de la instrucción correspondiente si la hay. Cuando el programa llega a su fin, se imprimirá por pantalla el AST del programa obtenido.

A diferencia del analizador sintáctico, cuando se encuentra un error el análisis del programa continua, con el objetivo de tratar de analizar la mayor parte de este posible. Los métodos ya no solo devolverán cierto o falso atendiendo a si se ha encontrado un error o no, sino que, en los casos donde sea necesario, devolverán además de este los atributos indicados en el ETDS ya definido.

Por último, vamos a explicar de forma rápida como hemos solucionado cada una de las restricciones definidas al inicio del apartado (omitiendo 1, 2 y 3, que ya se han visto en el apartado de la tabla de símbolos).

Para la restricción 4 hemos de realizar una conversión implícita de booleanos en números para realizar operaciones aritméticas, como sucede en la regla 3. Para ello, en reglas como la 32 del no terminal resto termino, donde, independientemente del tipo de los operandos para una operación de multiplicación, los operandos se tratan como números y devuelven un árbol de tipo número.

```
def resto_term_32(self,arbol,tipo):
    correcto, operador = self.opmult()
    if not(correcto): return False, NodoVacio(self.token.linea), "numero"
    correcto1, arbol1, tipo1 = self.factor()
    correcto2,arbol2,_ = self.resto_term(arbol1,tipo1)
    correcto = correcto and correcto1 and correcto2
    return correcto,NodoAritmetico(arbol,arbol2,self.token.linea,operador) ,"numero"
```

Imagen 37: regla donde se evalúa la restricción 4

Cabe destacar que esto tendrá lugar de la misma forma en todas las reglas que usen operadores de tipo suma multiplicación o relacionales (en este último solo es imprescindible en el caso de que uno de los operandos sea de tipo numérico).

En lo relativo a la restricción 5, que hace referencia a que no existe conversión de entero en booleano, podemos observar que, para nuestra implementación, tanto las operaciones de booleanos como la asignación no usan conversión de tipos, es decir, sus parámetros son siempre de tipo booleano, y, en el caso de no serlo, devuelven un nodo vacío. Esto lo podemos observar en el método equivalente a la regla 33, como se ve en la imagen inferior:

```
def resto_term_33(self,arbol,tipo):
    correcto, operador = self.y()
    if not(correcto): return False, NodoVacio(self.token.linea), "booleano"
    correcto1,arbol1,tipo1 = self.factor()
    correcto2, arbol2,tipo2 = self.resto_term(arbol1,tipo1)
    correcto = correcto1 and correcto2
    if(tipo=="booleano" and tipo2 == "booleano"):
        return correcto, NodoOPBool(arbol,arbol2,self.token.linea,operador), "booleano"
    return correcto, NodoVacio(self.token.linea), "numero"
```

Imagen 38: regla donde se evalúa la restricción 5

En cuanto a la restricción 6, que indica que para la instrucción lee el argumento deberá de ser un identificador de tipo entero o real, esto lo tratamos en el método correspondiente a la regla 21, donde hacemos uso de la tabla de símbolos. El fragmento de código que nos permite comprobar esta restricción es el siguiente:

```
if(self.esta_tabla_simbolos(id)):
    tipo = self.tabla_simbolos[id][1]
    if(tipo !="numero"):
        return False,NodoVacio(self.token.linea)
else:
    return False,NodoVacio(self.token.linea)
```

Imagen 39: regla donde se evalúa la restricción 6

7. PRUEBAS DE NUESTRA IMPLEMENTACIÓN

En este caso, hemos usado un total de 4 ficheros para probar el funcionamiento de esta fase del análisis y tratar de detectar y corregir la mayor cantidad de errores de nuestra implementación.

- Fichero de texto 1: lo podemos encontrar con el nombre entrada1_anasem.txt, con el que buscamos probar la declaración de variables correcta y la generación de un primer AST. Su contenido es el siguiente:

```
PROGRAMA p1;
VAR i,x:ENTERO;
INICIO
      i:=0.0;
      MIENTRAS (i<>5) HACER
            INICIO
            x:=i*4;
            ESCRIBE(x);
      FIN;
FIN.
```

Imagen 40: contenido del fichero entrada1_anasem.txt

Este para la declaración de variables nos da el siguiente resultado, que es el esperado:

```
Lista de identifcadores encontrada: ['i', 'x']
El tipo encontrado es  entero
Insercion en la tabla de simbolos para la variable  i  con exito
Insercion en la tabla de simbolos para la variable  x  con exito
```

Imagen 41: salida de la declaración de variables del fichero entrada1_anasem.txt

En cuanto al resultado del árbol obtenido, este es el siguiente:

```
FIN DEL ARCHIVO
La cantidad de errores encontrados es de  0
Se llega al final del analisis
El AST obtenido es el siguiente:
( "Compuesta"
 ( "Asignacion"  "linea: 4" i ( "Real" "valor: 0.0" "tipo: numero" "linea: 4" )
)
( "Mientras" "linea: 9" ( "Comparacion" "op: <>" "tipo: booleano" "linea: 5"
 ( "AccesoVariable" "v: i" "linea: 5" )
 ( "Real" "valor: 5" "tipo: numero" "linea: 5" )
)
 ( "Compuesta"
 ( "Asignacion"  "linea: 7" x ( "Aritmetica" "op: *" "tipo: numero" "linea: 7"
 ( "AccesoVariable" "v: i" "linea: 7" )
 ( "Real" "valor: 4" "tipo: numero" "linea: 7" )
)
)
( "Escribe" "linea: 8" ( "AccesoVariable" "v: x" "linea: 8" ) )
)
 )
)
Analisis semántico SATISFACTORIO. Fichero : entrada1_anasem.txt CORRECTO
```

Imagen 42: AST obtenido del fichero entrada1_anasem.txt

- Fichero de texto 2: lo podemos encontrar con el nombre entrada2_anasem.txt. Este se trata del fichero anterior al que le añadimos una declaración de otra variable, pero de tipo booleano y una mayor complejidad en las instrucciones para poder detectar fallos en la generación del árbol. En contenido de este es el siguiente:

```
PROGRAMA p1;
VAR i,x:ENTERO;z:BOOLEANO;
INICIO
      x:=0.0;
      MIENTRAS (i<>5) HACER
            INICIO
            x:=i*4;
            ESCRIBE(x);
            FIN;
      INICIO
            SI (x < 0) ENTONCES LEE (i) SINO z:=(z O FALSO);
      FIN;
FIN.
```

Imagen 42: contenido del fichero entrada2_anasem.txt

La salida, al igual que con el anterior, es la esperada, siendo esta la siguiente:

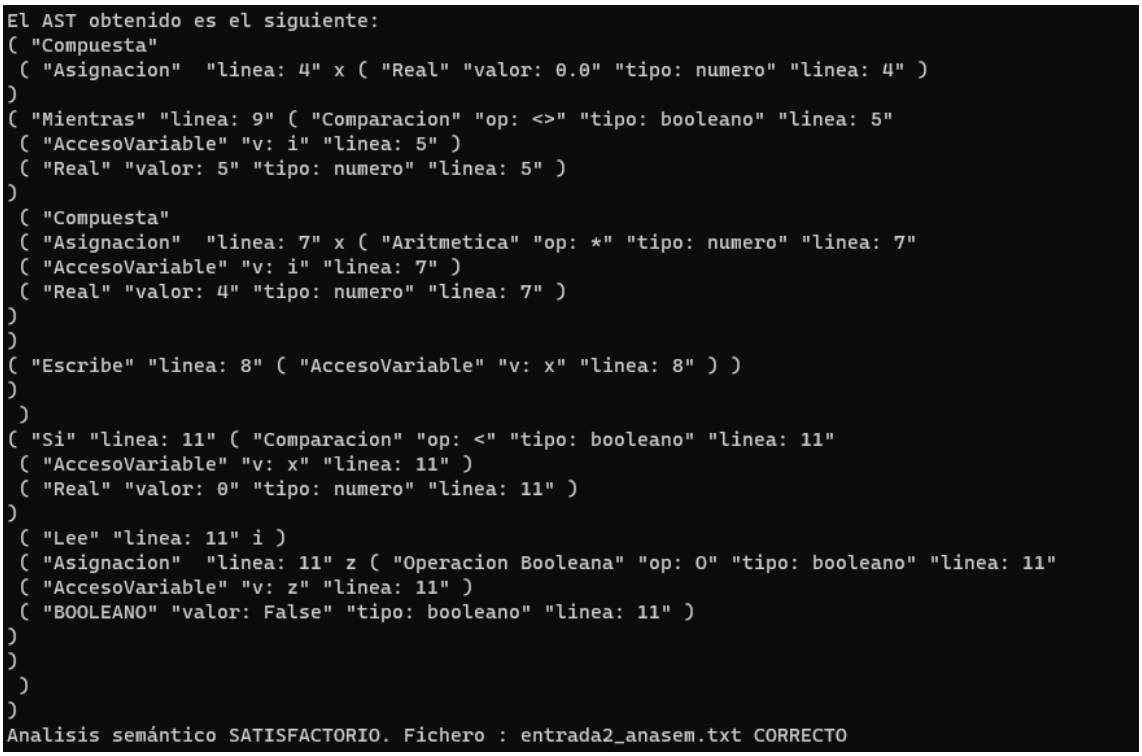

Imagen 43: salida del fichero entrada2_anasem.txt

De igual forma, la declaración de variables se realiza de forma correcta:

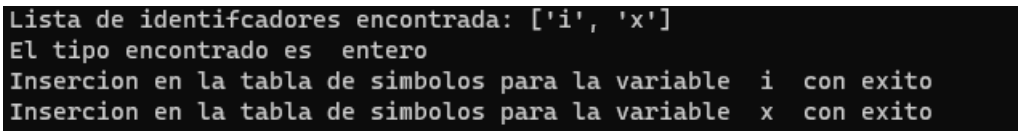

Imagen 44: declaración 1 de variables en el fichero entrada2_anasem.txt

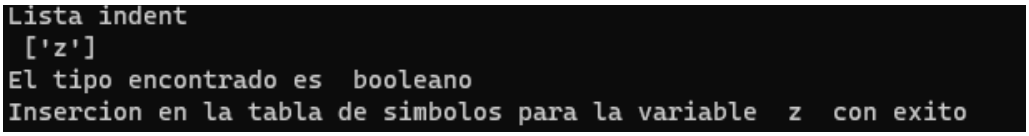

Imagen 45: declaración 2 de variables en el fichero entrada2_anasem.txt

- Fichero de texto 3: lo podemos encontrar con el nombre entrada3_anasem.txt. En este caso, pasamos a comprobar el funcionamiento del programa en caso de errores. Para ello, eliminamos del anterior fichero la declaración de la variable x para comprobar que el programa detecte un error en el caso de usar alguna instrucción con ella. Además, añadimos una instrucción que haga uso de argumentos numéricos para una operación booleana para comprobar que también obtenemos un error, en este caso devolviendo un nodoVacio. El contenido es el siguiente:

```
PROGRAMA p1;
VAR i:ENTERO;z:BOOLEANO;
INICIO
      x:=0.0;
      MIENTRAS (i<>5) HACER
            INICIO
            x:=i*4;
            ESCRIBE(x);
            FIN;
      INICIO
            SI (i < 0) ENTONCES LEE (i) SINO z:=(i O FALSO);
      FIN;
FIN.
```

Imagen 44: contenido del fichero entrada3_anasem.txt

Para el uso de la variable sin declarar, el error emitido por pantalla es el siguiente:

```
Uso de variable sin declarar:  x
```

Imagen 45: error en la variable en el análisis del fichero entrada3_anasem.txt

Dado que el programa esta formado por un nodo compuesto que contiene 3 instrucciones, y estas a su vez todas contienen errores (uso de variable sin declarar en las 2 primeras u uso de variable numérica para una operación booleana en la última), el resultado devuelto es un nodo vacío:

```
Se llega al final del analisis
El AST obtenido es el siguiente:
( "Nodo Vacio linea: 13" )
Analisis semántico CON ERRORES. Fichero : entrada3_anasem.txt ERRONEO
```

Imagen 46: nodo devuelto por el análisis semántico

- Fichero de texto 4: lo podemos encontrar con el nombre entrada4_anasem.txt. En este probaremos alguna restricción definida aun no probada, como el uso de una operación de lectura con argumento booleano, que nos devolverá un fallo, o testearemos el correcto funcionamiento de una operación aritmética con un parámetro numérico y otro booleano, que deberá de someterse a una conversión implícita de tipo, devolviendo un numérico. El fichero de prueba será el siguiente:

```
PROGRAMA p1;
VAR i,x:ENTERO;z:BOOLEANO;
INICIO
      x:=0.0;
      MIENTRAS (i<>5) HACER
            INICIO
            x:=i*4;
            LEE(z);
            FIN;
      INICIO
            SI (x < 0) ENTONCES LEE (i) SINO x:=(z + FALSO);
      FIN;
FIN.
```

Imagen 47: contenido del fichero entrada4_anasem.txt

Como podemos observar en la siguiente imagen, al haber un error en el while al usar lee una variable booleana, la sentencia no se añade al ATS. Además, podemos observar que la sentencia z+Falso nos devuelve un valor numérico, a pesar de ser sus argumentos ambos de tipo bool, ya que estos son los operandos de una operación aritmética.

```
Se llega al final del analisis
El AST obtenido es el siguiente:
( "Compuesta"
 ( "Asignacion"  "linea: 4" x ( "Real" "valor: 0.0" "tipo: numero" "linea: 4" )
)
( "Si" "linea: 11" ( "Comparacion" "op: <" "tipo: booleano" "linea: 11"
 ( "AccesoVariable" "v: x" "linea: 11" )
 ( "Real" "valor: 0" "tipo: numero" "linea: 11" )
)
 ( "Lee" "linea: 11" i )
 ( "Asignacion"  "linea: 11" x ( "Aritmetica" "op: +" "tipo: numero" "linea: 11"
 ( "AccesoVariable" "v: z" "linea: 11" )
 ( "BOOLEANO" "valor: False" "tipo: booleano" "linea: 11" )
)
)
 )
)
Analisis semántico CON ERRORES. Fichero : entrada4_anasem.txt ERRONEO
```

Imagen 48: ATS devuelto por el fichero entrada4_anasem.txt

- Fichero de texto 5: una ultima prueba que hemos considerado necesaria es, para el último condicional, cambiar su segunda expresión por x:=((z + FALSO) O FALSO), de tal forma que podamos observar que no se realiza la conversión implícita para realizar la operación O. Este fichero, por lo tanto, tiene la siguiente forma:

```
PROGRAMA p1;
VAR i,x:ENTERO;z:BOOLEANO;
INICIO
      x:=0.0;
      MIENTRAS (i<>5) HACER
            INICIO
            x:=i*4;
            LEE(z);
            FIN;
      INICIO
            SI (x < 0) ENTONCES LEE (i) SINO x:=((z + FALSO) O FALSO);
      FIN;
FIN.
```

Imagen 49: contenido del fichero entrada5_anasem.txt

Y su resultado al ejecutarlo, como era de esperar, es el siguiente:

```
Se llega al final del analisis
El AST obtenido es el siguiente:
( "Asignacion"  "linea: 4" x ( "Real" "valor: 0.0" "tipo: numero" "linea: 4" )
)
Analisis semántico CON ERRORES. Fichero : entrada5_anasem.txt ERRONEO
```

Imagen 50: ATS devuelto por el fichero entrada5_anasem.txt

Solo se inserta el nodo de la asignación, es decir, la primera, porque por un lado en el bucle existe un error, y por lo tanto no se inserta en el nodo compuesto, y por otro la segunda expresión del condicional contiene otro, ya que en la operación booleana O el primer operando es de tipo numérico. Dado que en el nodo compuesto la cantidad de nodos que lo componen es de uno, se devuelte este único nodo en lugar de instanciar uno compuesto.

CONCLUSIONES

En general, sí que nos parece que la realización de las prácticas y de la memoria es útil para demostrar y reforzar todos los conocimientos aprendidos a lo largo de la asignatura, por lo que sí que recomendaríamos que se siguieran llevando a cabo a lo largo de los próximos cursos. Si tuviéramos que ponerle una pega, esta sería que en la última práctica se da demasiada libertad en la implementación, en especial en las restricciones y debido a las posibles interpretaciones diferentes del enunciado. Esto nos ha preocupado de forma especial al no haber tenido una entrevista donde se pudiera probar el código de este apartado.

Por último, pensamos que las practicas previas a la de la memoria no nos parecen tan interesantes como esta, y estas sí que las sustituiríamos por una parte más de esta práctica donde haya que hacer algo relacionado con el último tema, si es que tiene sentido hacer una implementación de un generador de código.