Capítulo 2. Analizador Sintáctico

1 Objetivo

El objetivo del Analizador Sintáctico es identificar las estructuras (definiciones, sentencias, expresiones, etc.) que haya en el fichero de entrada y comprobar que todas ellas están correctamente formadas.

Por ejemplo, supóngase la siguiente entrada:

```
DATA
        float precio;
        int ancho;
CODE
        ancho = 25 * (2 + 1);
        print ancho;

        precio = 5.0;
        print precio / 2.0;
```

En esta entrada, al ejecutar el analizador sintáctico, este indicará que el programa pertenece al lenguaje:

```
El programa se ha compilado correctamente.
```

Sin embargo, si alguna estructura no está bien construida (o bien faltan/sobran elementos, o bien no están en el orden adecuado) entonces el sintáctico notificará el error. Suponiendo la siguiente entrada errónea:

```
DATA
        precio;    // Falta el tipo de la variable
        int ancho;
CODE
```

Se obtendría el siguiente resultado notificando el error:

```
line 2:1 missing 'CODE' at 'precio'
line 2:7 mismatched input ';' expecting {'CODE', 'float', 'int'}
line 3:10 mismatched input ';' expecting {'CODE', 'float', 'int'}
```

2 Trabajo autónomo del alumno

Diseñar e implementar esta fase *antes de seguir leyendo* el resto del documento. Una vez hecho, comparar la solución del alumno con la planteada a continuación.

3 Solución

3.1 Diseño. Creación de la Especificación

3.1.1 Extracción de Requisitos

Como en todo capítulo, se comenzará extrayendo los requisitos propios de la fase actual del compilador.

Se incluyen a continuación los requisitos de *Descripción del Lenguaje.pdf* que corresponden a la fase de análisis sintáctico:

- La sección *DATA* deberá aparecer obligatoriamente antes de la sección *CODE* y cada una de ellas solo puede aparecer una vez.
- En la sección *DATA* se realizan las definiciones de variables (no puede haber definiciones en la sección *CODE*).
- No es obligatorio que se definan variables, pero en cualquier caso tiene que aparecer la palabra reservada *DATA*.
- Las variables solo pueden ser de tipo entero o tipo real (*float*).
- En cada definición habrá una sola variable (no se permite "*int a,b;*")
- En la sección *CODE* aparecen las sentencias del programa (no puede haber sentencias en la sección *DATA*).
- Un programa puede no tener ninguna sentencia, pero en cualquier caso es obligatorio que aparezca la palabra reservada *CODE*.
- En la sección *CODE* puede haber dos tipos de sentencias: escritura (*print*) y asignación.
- Las expresiones podrán estar formadas por literales enteros, literales reales y variables combinados mediante operadores aritméticos (suma, resta, multiplicación y división).
- Se podrán agrupar expresiones mediante paréntesis.

Otro requisito que se observa en el ejemplo de *programa.txt* es que todas las sentencias acaban en punto y coma (cosa que en la descripción en lenguaje natural no se mencionaba).

```
DATA
        float precio;
        int ancho;
        int alto;
        float total;

CODE
        precio = 9.95;
        total = (precio – 3.0) * 1.18;
        print total;

        ancho = 10; alto = 20;
        print 0 - ancho * alto / 2;
```

3.1.2 Metalenguaje elegido

En esta fase se requiere un metalenguaje que permita describir de manera precisa las estructuras que forman un programa válido, qué elementos las componen y en qué orden.

Se usará para ello una Gramática Libre de Contexto (GLC). En una Gramática, de manera muy resumida, se definen las estructuras del lenguaje (mediante símbolos no-terminales) y se indica la forma de cada una de ellas mediante reglas de producción. Las reglas que definen una estructura siguen tres construcciones básicas:

- **Secuencias**. Indican el orden el que deben aparecer los símbolos de la estructura.
- **Listas**. Indican que la estructura se forma repitiendo otra estructura.
- **Composiciones**. En una composición se definen las formas atómicas (básicas) de la estructura para luego formar estructuras mayores que se apoyan en las primeras.

3.1.3 Especificación en el Metalenguaje

Los requisitos para el Analizador Sintáctico extraídos en el apartado anterior, expresados en una GLC en BNF son:

```
// En BNF
programa: DATA variables CODE sentencias

variables:
   | variables variable

variable:  tipo IDENT ';'

tipo: INT
   | REAL

sentencias:
   | sentencias sentencia

sentencia: PRINT expr ';'
            | expr '=' expr ';'

expr: IDENT
   | LITERALINT
   | LITERALREAL
   | expr '+' expr
   | expr '-' expr
   | expr '*' expr
   | expr '/' expr
   | '(' expr ')'
```

Se ha definido *programa* como una *secuencia* que indica que lo primero son las variables y luego las sentencias. Otros ejemplos de la construcción *secuencia* son los no-terminales *variable*, *tipo* y *sentencia*, que indican respectivamente cómo se define una variable y cómo se ordenan una escritura y una asignación.

Los no-terminales *variables* y *sentencias* se han definido mediante *listas* (en este caso mediante el patrón de cero o más elementos sin separadores).

Por último, las *expresiones* son una *composición* en la que se definen sus casos atómicos (los identificadores y los literales) y luego se define cómo se forman otras expresiones mayores en función de las anteriores.

A la hora de construir una gramática es *muy recomendable* que a la hora de crear cada regla se plantee ésta como una de las tres construcciones básicas. En caso contrario, es fácil acabar con reglas heterodoxas (como por ejemplo que sean a la vez secuencia y lista) que complican la gramática innecesariamente (y también su implementación).

Dado que ANTLR lo permite, la gramática anterior BNF se puede expresar en EBNF simplificando así alguna de las reglas (concretamente las listas):

```
// En EBNF

programa: 'DATA' variable* 'CODE' sentencia*;

variable: tipo IDENT ';';

tipo: 'float' | 'int';

sentencia: 'print' expr ';'
         | expr '=' expr ';';

expr     : expr ('+' | '-') expr
         | expr ('*' | '/') expr
         | '(' expr ')'
         | IDENT
         | INT_CONSTANT
         | REAL_CONSTANT
         ;
```

Para acabar, nótese que los requisitos sintácticos expresados en el metalenguaje no solo son más precisos, sino que además son mucho más concisos; solo hay que comparar lo que ocupan los requisitos en la GLC y lo que ocupaban en lenguaje natural en el apartado 3.1.1.

3.1.4 Aclaración sobre la sentencia Asignación

Nótese cómo se ha definido la asignación en la solución anterior:

```
sentencia: expr '=' expr ';'
```

En realidad, en este lenguaje tan sencillo, hubiera sido más correcto definir la asignación de la siguiente manera, ya que lo único que puede aparecer a la izquierda de una asignación es una variable:

```
sentencia: 'IDENT' '=' expr ';'
```

El motivo por el que se ha puesto como primer hijo una expresión en lugar de una variable es meramente didáctico. Aunque en este lenguaje no hace falta, así es como realmente habría que hacerlo a poco que el lenguaje se amplíe (como será en el caso de la práctica del alumno), ya que, a la izquierda de una asignación, además de una variable, pueden aparecer

accesos a arrays, estructuras, etc. Se muestra así cómo sería una versión más real del modelado de una asignación lo que permitirá, en posteriores capítulos explicar aspectos adicionales.

3.2 Implementación

3.2.1 Fichero de Especificación de ANTLR

Ahora hay que implementar una función que reciba tokens del Analizador Léxico y determine, mediante la implementación de las reglas de producción, si éstos forman estructuras válidas del lenguaje.

Aunque se podría hacer a mano, para acelerar el proceso se utilizará la herramienta *ANTLR*. Para ello hay que escribir en un fichero la gramática que describe nuestro lenguaje. En la carpeta *parser* está el fichero *Grammar.g4*, el cual ahora mismo sólo reconoce enteros. Habría que actualizarlo con la nueva especificación. Esta puede ser la versión en BNF:

```
grammar Grammar;
import Lexicon;

start: 'DATA' variables 'CODE' sentences EOF;

variables:    | variables variable;
variable: tipo IDENT ';';

tipo: 'float' | 'int';

sentences:    | sentences sentence;
sentence: 'print' expr ';' | expr '=' expr ';';

expr       : expr '+' expr
           : expr '-' expr
           : expr '*' expr
           | expr '/' expr
           | '(' expr ')'
           | IDENT
           | INT_CONSTANT
           | REAL_CONSTANT
           ;
```

O la versión en EBNF:

```
grammar Grammar;
import Lexicon;

start: 'DATA' variable* 'CODE' sentence* EOF;

variable: tipo IDENT ';';

tipo: 'float' | 'int';

sentence: 'print' expr ';'
          | expr '=' expr ';';

expr       : expr ('*' | '/') expr
           | expr ('+' | '-') expr
           | '(' expr ')'
           | IDENT
```

```
        | INT_CONSTANT
        | REAL_CONSTANT
        ;
```

Nótese que, en estas soluciones, se hace uso del hecho de que si un token está formado por un solo lexema (*print*, *DATA*, operadores, etc.), *no es necesario definirlo* en el fichero léxico (*Lexicon.g4*), ya que se puede definir directamente en *Grammar.g4* usando las comillas simples. Por esto, el fichero *Lexicon.g4* quedaría así:

```
lexer grammar Lexicon;

INT_CONSTANT:           [0-9]+;
REAL_CONSTANT:          [0-9]+ '.' [0-9]+;
IDENT:                  [a-zA-Z][a-zA-Z0-9_]*;

LINE_COMMENT:           '//' .*? ('\n'|EOF)  -> skip;
MULTILINE_COMMENT:      '/*' .*? '*/'        -> skip;

WHITESPACE: [ \t\r\n]+ -> skip;
```

3.2.2 Uso de ANTLR

Una vez modificados los ficheros *Grammar.g4* (con la versión BNF o la EBNF − a elección del alumno) y *Lexicon.g4*, quedaría solamente usar ANTLR para generar el código Java correspondiente. Para ello, hay que ejecutar ANTLR.bat:

```
antlr.bat
```

Aparecerán entonces en la carpeta *parser* los ficheros *GrammarLexer.java* (el analizador léxico) y *GrammarParser.java* (el analizador sintáctico).

Nota. Hay veces en que *eclipse* no detecta el cambio en un fichero generado por una herramienta y por tanto seguirá compilando la versión antigua del mismo. Si esto ocurre, debe seleccionarse el proyecto en la ventana *Package Explorer* y pulsar F5 para actualizar.

4 Ejecución

Se crea en *source.txt* el siguiente programa para probar el analizador:

```
DATA
        float precio;
        int ancho;
CODE
        ancho = 25 * (2 + 1);
        print ancho;

        precio = 5.0;
        print precio / 2.0;
```

Para probar el analizador sintáctico basta ahora con ejecutar la clase *main.Main*. Este seguirá sacando el mensaje esperado (ya que aún no se construye el AST):

```
El AST no ha sido creado.
```

Sin embargo, se puede comprobar que está reconociendo la entrada como válida (ya que no da mensajes de error).

Se recomienda introducir cambios en el fichero *source.txt* para comprobar cómo el analizador detecta los programas que tengan errores sintácticos.

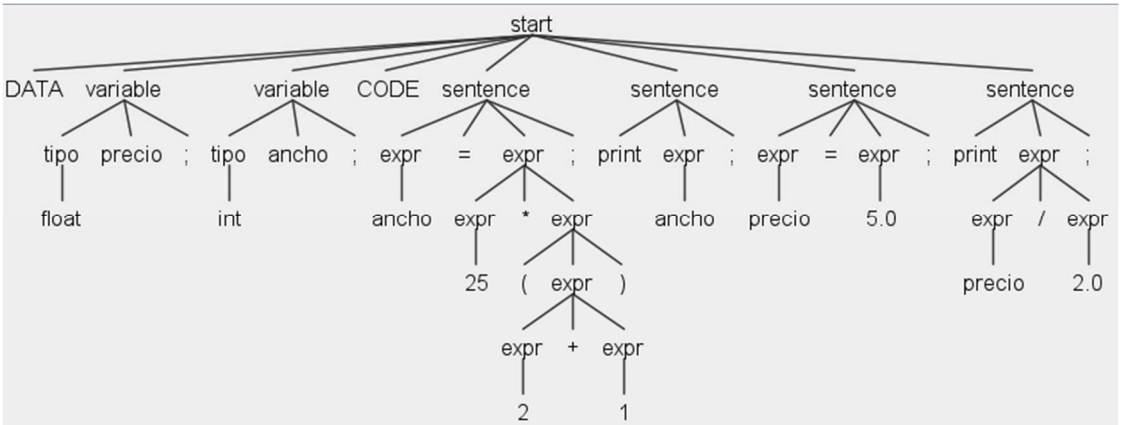
5 Traza del Análisis Sintáctico

Como se ha podido comprobar, el sintáctico se está comportando ahora mismo como una caja negra. Sólo se sabe si la entrada es válida o no, pero no cómo lo ha hecho.

Para saber el árbol sintáctico que se ha creado para la entrada, es decir, las transformaciones que se han aplicado para llegar desde el símbolo inicial de la gramática hasta la cadena de entrada, se puede utilizar la utilidad *TestRig* de ANTLR. Para invocarla, hay preparado un fichero BAT:

```
gui.bat source.txt
```

Esto muestra el siguiente árbol sintáctico (concreto):



En caso de que la entrada no pertenezca al lenguaje, nos indica qué parte de la cadena no se corresponde con ninguna estructura del lenguaje. Supongamos, por ejemplo, que se omite el punto y coma de en la definición de la variable *ancho*:

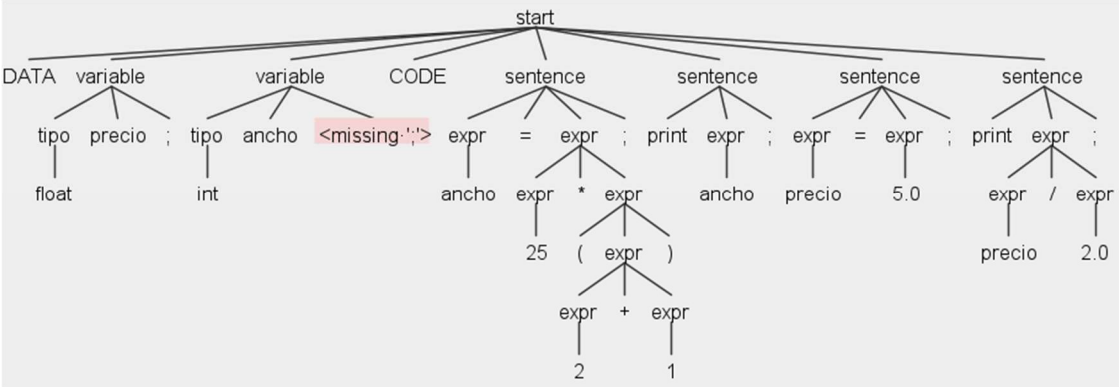
```
DATA
  float precio;
  int ancho   // <-- Aquí falta ';'
CODE
  ancho = 25 * (2 + 1);
  print ancho;

  precio = 5.0;
  print precio / 2.0;
```

Ejecutando el main, nuestro compilador detectará el error. Si, además, queremos tener más información de por qué la entrada es errónea, se puede usar de nuevo TestRig:

```
gui.bat source.txt
```

En este caso, muestra en rojo que la definición de dicha variable no ha podido reconocerse correctamente por faltar un token (el punto y coma).



6 Resumen de Cambios

Fichero	Acción	Descripción
Grammar.g4	Modificado	Se ha añadido la gramática del lenguaje
Lexicon.g4	Modificado	Se han pasado los tokens de un solo lexema al fichero Grammar.g4
source.txt	Modificado	Se ha creado una prueba que tuviera todas las construcciones sintácticas del lenguaje
GrammarParser.java	Generado	Implementación del Analizador Sintáctico. Creado con *ANTLR* a partir de *Grammar.g4*
GrammarLexer.java	Generado	Implementación del Analizador Léxico. Creado con *JFlex* a partir de *Lexicon.g4*