

Capítulo 2. Analizador Sintáctico

1 Objetivo

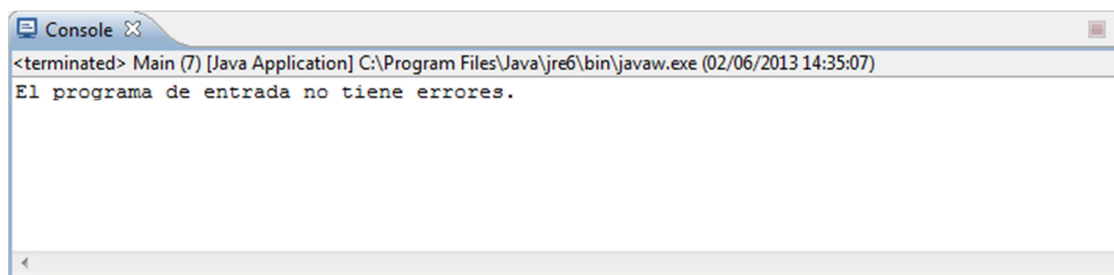
El objetivo del Analizador Sintáctico es identificar las estructuras (definiciones, sentencias, expresiones, etc.) que haya en el fichero de entrada y comprobar que todas ellas están correctamente formadas.

Por ejemplo ante una entrada como:

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

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

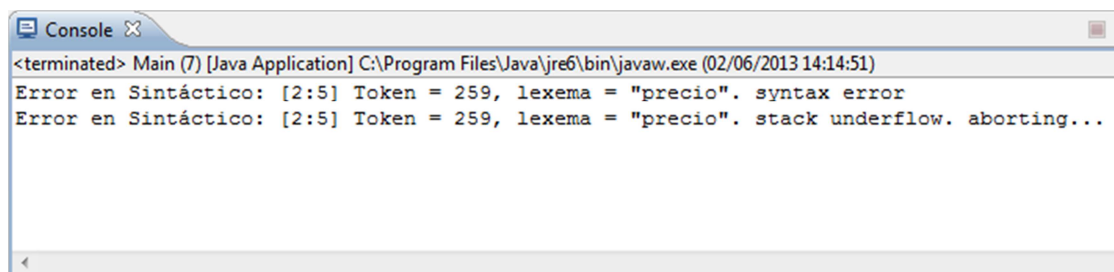
El analizador sintáctico indicará que el programa pertenece al lenguaje:



```
Console X
<terminated> Main (7) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (02/06/2013 14:35:07)
El programa de entrada no tiene errores.
```

Sin embargo, si alguna estructura no está bien construida (o bien faltan o sobran elementos o bien no están en el orden adecuado) entonces el sintáctico notificará el error:

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



```
Console X
<terminated> Main (7) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (02/06/2013 14:14:51)
Error en Sintáctico: [2:5] Token = 259, lexema = "precio". syntax error
Error en Sintáctico: [2:5] Token = 259, lexema = "precio". stack underflow. aborting...
```

2 Solución

2.1 Diseño. Creación de la Especificación

2.1.1 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 *Introducción.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*). ~~Las variables de tipo entero ocupan 2 bytes y las reales 4.~~
- 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 y asignación.
- La escritura (*print*) puede ser tanto de expresiones enteras como reales.
- 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;
```

2.1.2 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 otras estructuras.
- **Composiciones.** En una composición se definen las formas atómicas (básicas) de la estructura para luego formar estructuras mayores que se apoyan en las primeras.

2.1.3 Especificación

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

```
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 más 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).

Para acabar, nótese que los requisitos sintácticos expresados en el metalenguaje no solo son más precisos sino que además es mucho más conciso; solo hay que comparar lo que ocupan los requisitos en la GLC y lo que ocupaban en lenguaje natural en el apartado 2.1.1.

2.2 Implementación

2.2.1 Fichero de Especificación de BYaccJ

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 *BYaccJ*. Para ello hay que escribir en un fichero la gramática que describe nuestro lenguaje. En la carpeta *sintactico* está el fichero *sintac.y* el cual contiene un esqueleto de una especificación de BYaccJ con todo el código habitual de cualquier analizador sintáctico. Solo habría que añadir la GLC de nuestro lenguaje a la segunda sección de dicho fichero (parte con fondo gris):

```
/* No es necesario modificar esta sección ----- */
%{
package sintactico;

import java.io.*;
import java.util.*;
import ast.*;
import main.*;
%}

/* Precedencias aquí ----- */
%left '+' '-'
%left '*' '/'
%%

/* Añadir las reglas en esta sección ----- */

programa: 'DATA' variables 'CODE' sentencias

//-----
variables:
    | variables variable

variable: tipo 'IDENT' ';'

tipo: 'INT'
    | 'REAL'
```

```
//-----
sentencias:
| sentencias sentencia

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

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

%%
/* No es necesario modificar esta sección ----- */

public Parser(Yylex lex, GestorErrores gestor, boolean debug) {
    this(debug);
    this.lex = lex;
    this.gestor = gestor;
}

// Métodos de acceso para el main -----
public int parse() { return yyparse(); }
public AST getAST() { return raiz; }

// Funciones requeridas por Yacc -----
void yyerror(String msg) {
    Token lastToken = (Token) yyval;
    gestor.error("Sintáctico", "Token = " + lastToken.getToken() + ", lexema = \""
+ lastToken.getLexeme() + "\". " + msg, lastToken.getStart());
}

int yylex() {
    try {
        int token = lex.yylex();
        yyval = new Token(token, lex.lexeme(), lex.line(), lex.column());
        return token;
    } catch (IOException e) {
        return -1;
    }
}

private Yylex lex;
private GestorErrores gestor;
private AST raiz;
```

Dado que la clase *Parser* generada por *BYaccJ* incluirá constantes para identificar a todos los tokens, ya **no será necesario** el fichero *Tokens.java* (el cual hacía esta función) y por tanto deberá ser borrado.

Además habrá que actualizar el fichero *lexico.l* para que utilice estas nuevas constantes:

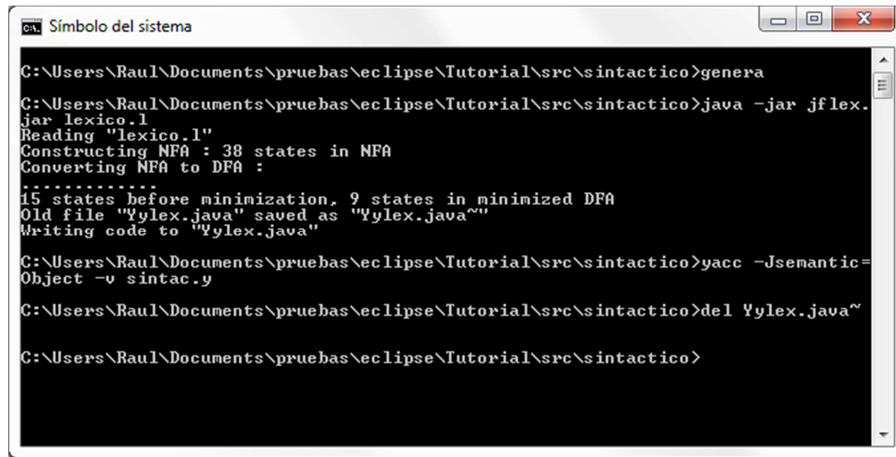
```
[+\\-*/;\\(\\)=] { return yytext().charAt(0); }

DATA { return Parser.DATA; }
CODE { return Parser.CODE; }
print { return Parser.PRINT; }
int { return Parser.INT; }
float { return Parser.REAL; }

[a-zA-ZñÑ][a-zA-Z0-9_ñÑ]* { return Parser.IDENT; }
[0-9]+ { return Parser.LITERALINT; }
[0-9]+\\. "[0-9]* { return Parser.LITERALREAL; }
```

2.2.2 Uso de BYaccJ

Una vez modificados los ficheros *sintac.y* y *lexico.l* quedaría solamente usar BYaccJ y JFlex para que cada uno de ellos genere el código Java correspondiente. Para ello hay que ir a la carpeta *sintactico* del proyecto y ejecutar *genera.bat*:



```
C:\Users\Raul\Documents\pruebas\eclipse\Tutorial\src\sintactico>genera
C:\Users\Raul\Documents\pruebas\eclipse\Tutorial\src\sintactico>java -jar jflex.
jar lexico.l
Reading "lexico.l"
Constructing NFA : 38 states in NFA
Converting NFA to DFA :
*****
15 states before minimization, 9 states in minimized DFA
Old file "Yylex.java" saved as "Yylex.java~"
Writing code to "Yylex.java"
C:\Users\Raul\Documents\pruebas\eclipse\Tutorial\src\sintactico>yacc -Jsemantic=
Object -v sintac.y
C:\Users\Raul\Documents\pruebas\eclipse\Tutorial\src\sintactico>del Yylex.java~
C:\Users\Raul\Documents\pruebas\eclipse\Tutorial\src\sintactico>
```

Aparecerá un nuevo fichero *Parser.java* generado por *BYaccJ*. Esta clase es la que implementará el Analizador Sintáctico. Además se habrá generado una versión actualizada de *Yylex.java* a partir de *léxico.l*.

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

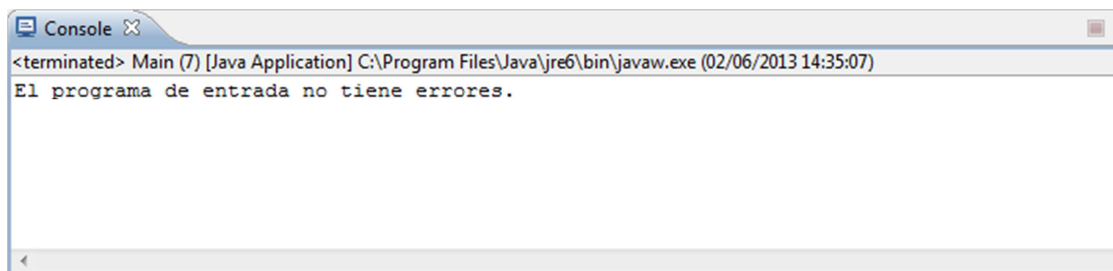
3 Ejecución

Se crea en *entrada.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*.



```
<terminated> Main (7) [Java Application] C:\Program Files\Java\jre6\bin\javaw.exe (02/06/2013 14:35:07)
El programa de entrada no tiene errores.
```

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

4 Resumen de Cambios

Fichero	Acción	Descripción
sintac.y	Modificado	Se ha añadido la gramática del lenguaje
Tokens.java	Borrado	La clase Parser ya realiza su función
lexico.l	Modificado	Se ha pasado de utilizar las constantes definidas en la clase Token a utilizar las definidas en la clase Parser
entrada.txt	Modificado	Se ha creado una prueba que tuviera todas las construcciones sintácticas del lenguaje
Parser.java	Generado	Implementación del Analizador Sintáctico. Creado con <i>BYaccJ</i> a partir de <i>sintac.y</i>
Yylex.java	Generado	Implementación del Analizador Léxico. Creado con <i>JFlex</i> a partir de <i>lexico.l</i>