PRÁCTICA DE PROCESADORES DEL LENGUAJE I

Curso 2017 – 2018

Entrega de Febrero

APELLIDOS Y NOMBRE: Manuel Rodríguez Sánchez

IDENTIFICADOR: mrodrigue212

CENTRO ASOCIADO MATRICULADO: Motril

CENTRO ASOCIADO DE LA SESIÓN DE CONTROL: Motril

MAIL DE CONTACTO: mrodrigue212@alumno.uned.es

GRUPO: B

1. Analizador léxico

A continuación se expone una descripción de los distintos bloques o sentencias modificadas o agregadas en el scanner.flex.

Se agregan las directivas siguientes:

%ignorecase: Esta opción hace que JFlex maneje todos los caracteres y cadenas en la especificación como si estuvieran tanto en mayúscula como en minúscula.

%full: esta opción hace que el escáner generado use un conjunto de caracteres de entrada de 8 bits (códigos de caracteres 0-255). Si se encuentra un carácter de entrada con un código superior a 255 en una entrada en tiempo de ejecución, el escáner arrojará una excepción *ArrayIndexOutofBoundsException*

%state COMENTARIO: declaración de un nuevo estado, además del estado por defecto YYINITIAL. A este estado se accederá cuando se detecte un comentario, todos los tokens que entren estando dentro de este estado, no se pasará al analizador sintáctico mediante *return*.

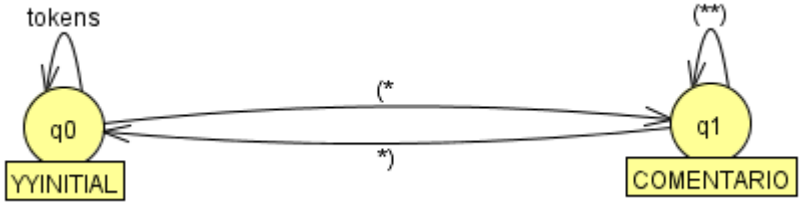

Ilustración 1 - Diagrama de estados

A continuación, se declara una variable de tipo entero, con el objetivo de almacenar el número de comentarios que se van abriendo (control de anidación de comentarios).

```
private int contadorComentario=0;
```

En siguiente lugar, con el objetivo de reducir código declaramos dos funciones tipo *Token*, una para la creación de tokens:

```
Token nuevoToken(int nToken){
      Token token = new Token (nToken);
      token.setLine (yyline + 1);
      token.setColumn (yycolumn + 1);
      token.setLexema (yytext ());
      return token;}
```

y la otra cuando encontramos una cadena de caracteres que se pasa como argumento de función u operación.

```
Token cadenaToken (int nToken, String cadenaSinComillas) {
      Token token = new Token(nToken);
      token.setLine (yyline + 1);
      token.setColumn (yycolumn + 1);
      token.setLexema (cadenaSinComillas);
      return token;}
```

a esta última función se le pasa el id del token y la cadena con las comillas (") eliminadas, es decir, le pasamos la cadena limpia. Esta operación se realiza en la macro {CADENA} (se detalla más adelante).

Al finalizar el análisis léxico, se ha de comprobar que los delimitadores de comentarios están correctamente balanceados. Esto lo hacemos mediante la directiva *%eofval{<código>%eofval}*, que permite definir un trozo de código para que se ejecute cuando llega a final de fichero.

```
%eofval{
{
      if(contadorComentario > 0){
            lexicalErrorManager.lexicalFatalError ("ERROR - Delimitadores de
            comentarios mal balanceados. Existe/n: "+contadorComentario+"
            comentario/s no cerrado/s.");
            contadorComentario = 0;
      }
      else return nuevoToken(sym.EOF);
}
%eofval}
```

Comprobamos que el contador de comentarios está a 0, si no fuera así, devolvemos error grave y detenemos el compilador.

Se declaran todas las macros que van a controlar entre otras cosas:

a. Los espacios en blanco: ESPACIO_BLANCO
b. Saltos de línea: SALTO_LINEA
c. Evitar los ceros a la izquierda en valores numéricos: ENTERO_TAL_CUAL
d. Evitar que los identificadores empiecen con un valor numérico: ID_ERRONEO
e. Control de las cadenas de caracteres entrecomilladas: {CADENA}

Después de la declaración de las macros, entramos en el estado YYINITIAL donde cada vez que entre uno de los tokens definidos a la izquierda, y que está entre comillas, se realice la acción correspondiente. Por lo general se devolverá un identificador de token al objeto *nuevoToken* mediante la sentencia:

```
{return nuevoToken([nombre_del_token]);}
```

Los casos especiales que podemos encontrarnos son:

"(*" Provoca que pasemos al estado COMENTARIO, aumentando con ello la variable contadorComentario, que nos permitirá controlar el balanceo de comentarios.

"*)" Al entrar este token se informa de un error, y no se pasa al analizador, pero se continúa la lectura de tokens. Hay que tener en cuenta, que este token solo ha de entrar cuando estamos en el estado COMENTARIO, donde aquí sí será útil para controlar el balanceo de comentarios anidados y con ello provocar o no la salida de este estado y por supuesto, la función que tiene como cierre de comentario.

{CADENA} La macro cadena se va a encargar de limpiar las cadenas que vengan entrecomilladas, con el objetivo de pasarle al analizador sintáctico la cadena sin las dobles comillas; esto se hace mediante la siguiente secuencia de sentencias en java:

```
String cadena=yytext();
cadena=cadena.substring(1,cadena.length()-1);
```

Una vez eliminadas, se devuelve usando el objeto *cadenaToken* creado para este fin, el identificador de token y la cadena resultante.

```
return cadenaToken(sym.CADENA,cadena);}
```

2. Analizador sintáctico

Comenzamos, desde el parser.cup, con la declaración de cada uno de los terminales que le llegarán en forma de token desde el analizador léxico.

Se hacen las correspondientes declaraciones, dejando las que había por defecto (program, Axiom); el resultado son un total de 45 terminales, 55 no terminales y 108 producciones declaradas.

Se establecen las reglas para las relaciones de precedencia siguientes:

```
precedence left PUNTO, PARENTESISABIERTO, PARENTESISCERRADO,
CORCHETEABIERTO, CORCHETECERRADO;
precedence left PRODUCTO;
precedence left PLUS, MINUS;

precedence rightNOT;
precedence left OR, MAYORQUE, IGUAL;
```

Comenzamos las reglas de producción iniciando éstas mediante:

```
axiom ::= modulUNED;
```

Se procura identificar de forma clara los distintos bloques con las producciones de cada una de las estructuras, y comportamiento de las instrucciones del lenguaje.

3. Conclusiones

Destacar que no importa la dificultad de este trabajo, lo importante es que a nivel teórico, el concepto de léxico y sintáctico en este contexto (e incluso fuera de él) aclara muchos aspectos y detalles del funcionamiento de un compilador, el cómo entran las cadenas de tokens o como se derivan los árboles gramaticales, y aquí he de destacar, que conforme se avanza en el trabajo las producciones gramaticales fluyen de una forma clara y sencilla.

A este trabajo se le han ido pasando los distintos test del grupo B; se ha probado con el programa *Listado 20* de la página 33 y 34 del enunciado de la práctica (se le ha llamado testCase09.muned). En las pruebas, se han incluido errores con el objetivo de ver si eran detectados tanto en el léxico como en el sintáctico, pasando estas pruebas favorablemente.

Como conclusión final, me ha resultado un trabajo altamente didáctico.

4. Gramática

A modo de ejemplo, se adjunta a continuación tres árboles de derivación de los tres primeros programas de los test de pruebas.

- **Árbol gramatical testCase01.muned**

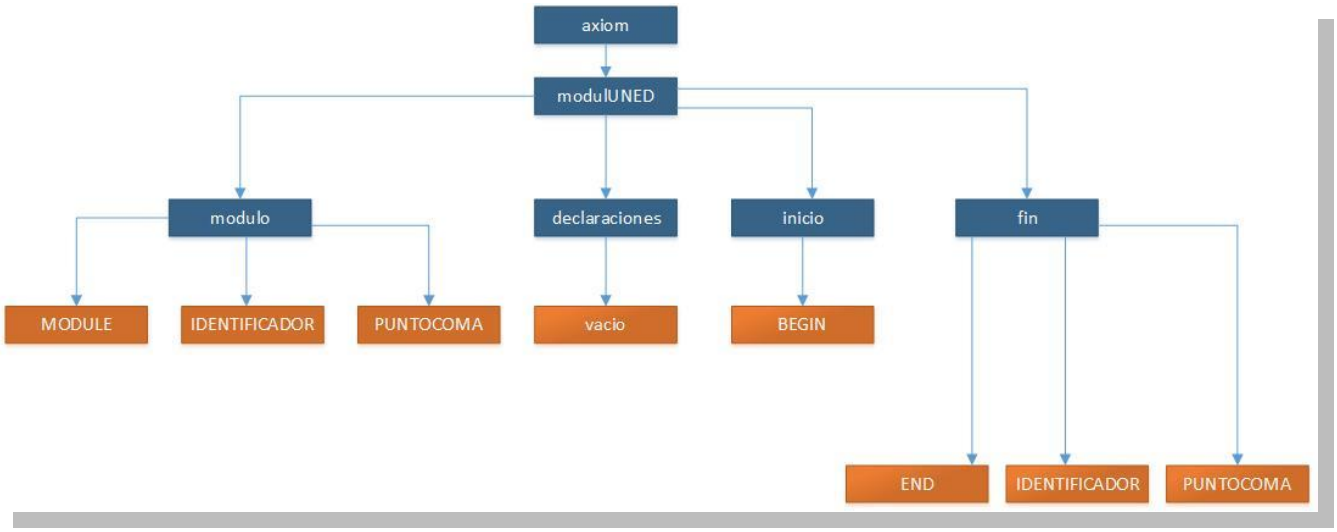

Ilustración 2 - Árbol derivación testCase01

- **Árbol gramatical testCase02.muned**

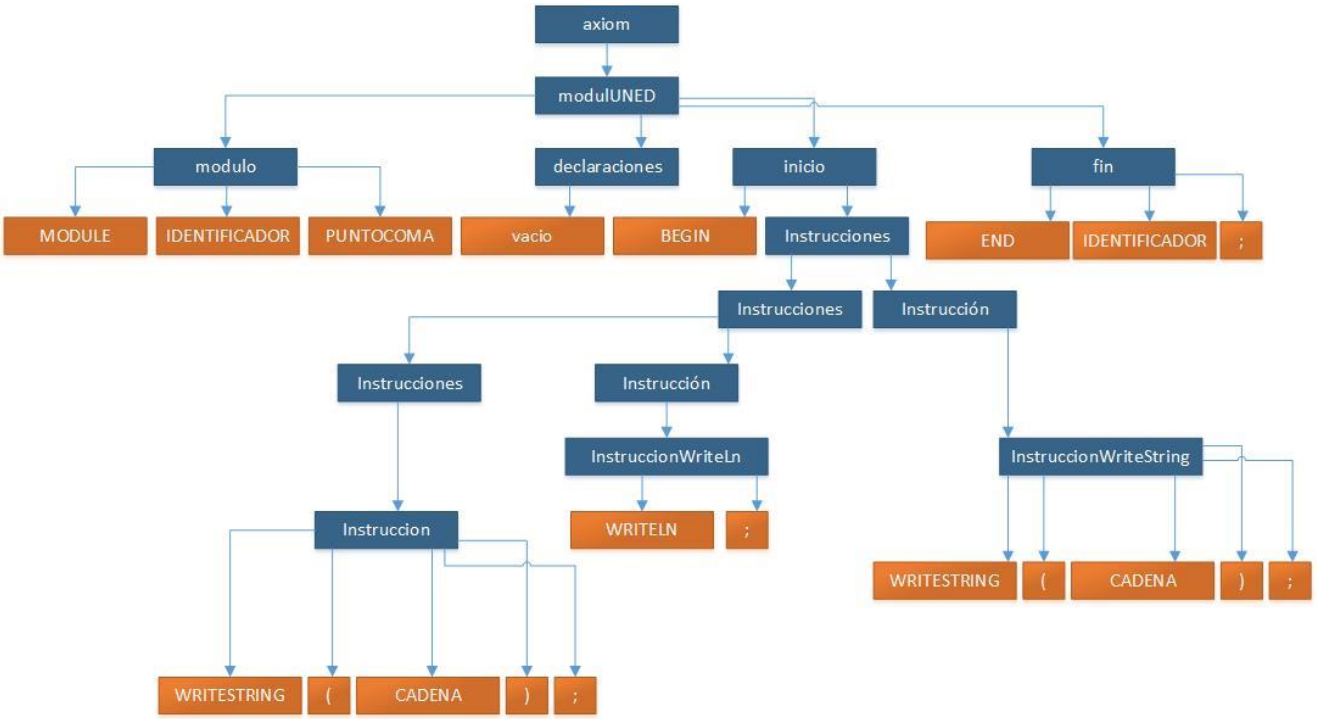

Ilustración 3 - Árbol derivación testCase02

- **Árbol gramatical testCase03.muned[1]**

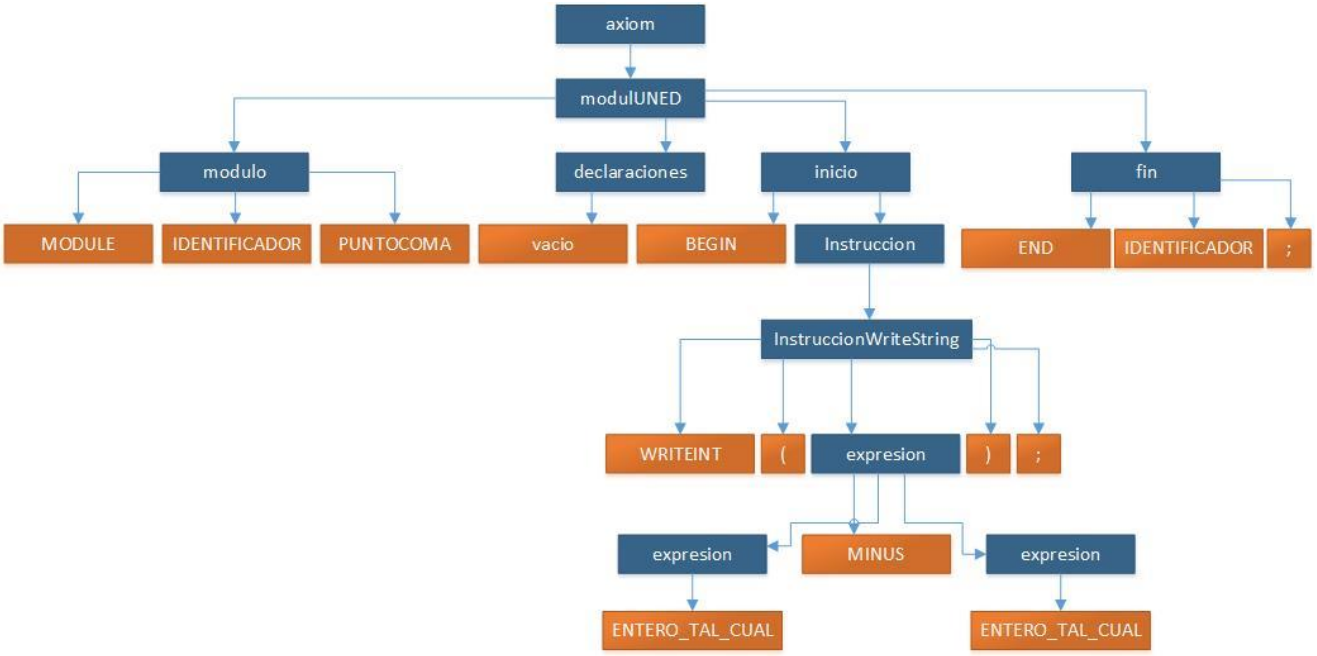

Ilustración 4 - Árbol derivación testCase03

[1] Por motivos de espacio, y dado que el objetivo de estos diagramas es mostrar el desarrollo de algunas reglas de producción, se han cambiado los nombres de algunos tokens por los símbolos que representan.