Capítulo 8

Generación de código

8.1 Visión general

Según el modelo de arquitectura de un compilador en el que éste se divide en *frontend* y *backend*, la etapa inicial traduce un programa fuente a una representación intermedia a partir de la cual la etapa final genera el código objeto, ya sea en forma de código máquina o ensamblador. Los detalles del lenguaje objeto se confinan en la etapa final, si esto es posible, lo que facilita la reutilización del *frontend* para crear otros compiladores del mismo lenguaje pero que generan código para otras plataformas. De esta forma, aunque a priori puede resultar más fácil traducir un programa fuente directamente al lenguaje objeto, las dos ventajas principales de utilizar una forma

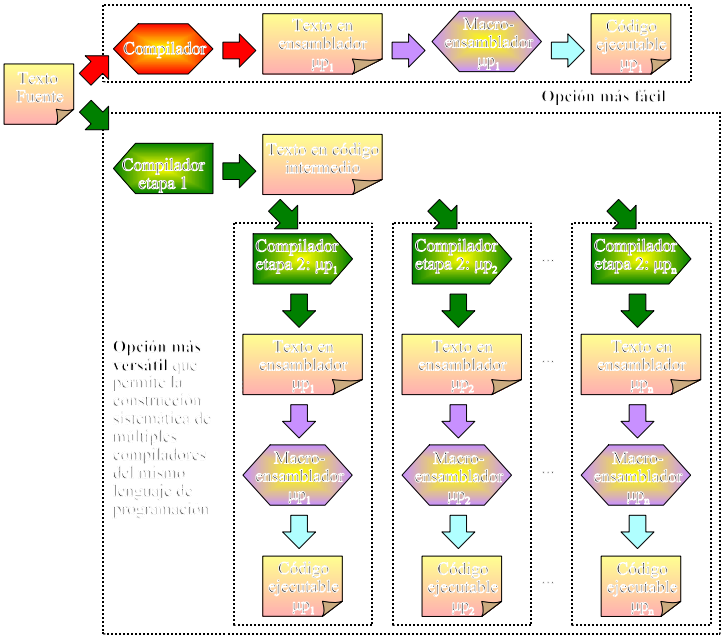


Figure 1 La construcción de un compilador mediante división en etapa *frontend* y etapa *backend* se realiza utilizando un código intermedio independiente de la máquina destino y en el que se codifica el programa fuente. El programa en código intermedio resultante es la salida de la etapa *frontend* y la entrada al *backend*

intermedia independiente de la máquina destino son:

- Se facilita la re-destinación, o sea, se puede crear un compilador para una máquina distinta uniendo una etapa final para la nueva máquina a una etapa inicial ya existente.
- Se puede aplicar a la representación intermedia un optimizador de código independiente de la máquina, lo que permite reutilizar también esta fase del compilador independientemente de la máquina destino.

La figura 8.1 muestra cómo el desarrollador debe optar por crear un compilador mediante un diseño versátil o mediante un diseño fácil. La elección final depende de la utilidad que se le vaya a dar al código fuente del compilador a corto o medio plazo, esto es, si se van a crear varios compiladores o uno sólo.

La filosofía versátil en la construcción de compiladores llega a su extremo en la implementación de lenguajes que son compilados y pseudointerpretados en ejecución. Esto quiere decir que en tiempo de compilación se genera un código máquina propio de un microprocesador virtual (llamado código-P en UCSD Pascal, *bytecodes* en Java, etc.) que, a su vez, se interpreta en tiempo de ejecución a través de lo que se llama **motor de ejecución**. A estos lenguajes no se les puede catalogar como interpretados ya que lo que se interpreta en tiempo de ejecución no es exactamente el programa fuente; pero tampoco se les puede considerar compilados del todo ya que lo que se genera en tiempo de compilación no es exactamente código máquina. Esta filosofía de diseño tiene la ventaja de que facilita la portabilidad de los programas. El lenguaje más actual que trabaja con esta filosofía es Java, que utiliza ficheros **.class**, en lugar de **.exe**. Los ficheros **.class** contienen *bytecodes* que se someten a una *Java Virtual Machine* (JVM) en tiempo de ejecución, para que los interprete. La JVM hace las veces de microprocesador virtual y los *bytecodes* hacen las veces de instrucciones máquina. Por supuesto, para poder ejecutar los *bytecodes* en diferentes plataformas es necesario que cada una de ellas posea una implementación adaptada de la JVM.

En este capítulo se muestra cómo se pueden utilizar los métodos de análisis dirigidos por la sintaxis para traducir un programa fuente a un programa destino equivalente escrito en código intermedio. Dado que las declaraciones de variables no generan código, los siguientes epígrafes se centran en las sentencias propias de un lenguaje de programación imperativo, especialmente asignaciones y cláusulas de control del flujo de ejecución. La generación de código intermedio se puede intercalar en el análisis sintáctico mediante las apropiadas acciones semánticas.

También es importante notar que, a veces, puede resultar interesante construir un traductor fuente-fuente en lugar de un compilador completo. Por ejemplo, si disponemos de un compilador de C y queremos construir un compilador de Pascal, puede resultar mucho más cómodo construir un programa que traduzca de Pascal a C, de manera que, una vez obtenido el programa C equivalente al de Pascal, bastará con compilar éste para obtener el resultado apetecido. Básicamente, las técnicas que se verán en este capítulo para generar código intermedio también pueden aplicarse a la generación de código de alto nivel.

8.2 Código de tercetos

Con el objetivo de facilitar la comprensión de la fase de generación de código, no nos centraremos en el código máquina puro de ningún microprocesador concreto, sino en un código intermedio cercano a cualquiera de ellos. Esta aproximación facilitará, además, la optimización del código generado.

Cada una de las instrucciones que podemos generar posee un máximo de cuatro apartados:

- Operando 1º (dirección de memoria donde se encuentra el primer operando).
- Operando 2º (dirección de memoria donde se encuentra el segundo operando).
- Operador (código de operación)
- Resultado (dirección de memoria donde albergar el resultado, o a la que saltar en caso de que se trate de una operación de salto).

Asumiremos que se permite tanto el direccionamiento directo como el inmediato, esto es, la dirección de un operando puede sustituirse por el operando en sí. A este tipo de instrucciones se las denomina códigos de 3 direcciones, tercetos, o códigos de máximo 3 operandos.

En esencia, los tercetos son muy parecidos a cualquier código ensamblador, existiendo operaciones para sumar, restar, etc. También existen instrucciones para controlar el flujo de ejecución, y pueden aparecer etiquetas simbólicas en medio del código con objeto de identificar el destino de los saltos.

No todas las instrucciones tienen porqué poseer exactamente todos los apartados mencionados. A modo introductorio, los tercetos que podemos usar son:

- Asignación binaria: **x := y op z**, donde **op** es una operación binaria aritmética o lógica. Aunque debe haber operaciones diferentes en función del tipo de datos con el que se trabaje (no es lo mismo sumar enteros que sumar reales), para facilitar nuestro estudio asumiremos que tan sólo disponemos del tipo entero.
- Asignación unaria: **x := op y**, donde **op** es una operación unaria. Las operaciones unarias principales incluyen el menos unario, la negación lógica, los operadores de desplazamiento de bits y los operadores de conversión de tipos.
- Asignación simple o copia: **x := y**, donde el valor de **y** se asigna a **x**.
- Salto incondicional: **goto etiqueta**.
- Saltos condicionales: **if x oprelacional y goto etiqueta**.

Como puede deducirse de los tercetos propuestos, las direcciones de memoria serán gestionadas de forma simbólica, esto es, a través de nombres en lugar de números. Por supuesto, para generar tercetos más cercanos a un código máquina general, cada uno de estos nombres debe ser traducido a una dirección de memoria única. Las direcciones de memoria se tomarían de forma consecutiva teniendo en cuenta el tamaño del tipo de datos que se supone alberga cada una de ellas; por

ejemplo, se puede asumir que un valor entero ocupa 16 bits, uno real ocupa 32 bits, un carácter ocupa 8 bits, etc.

Con los tercetos anteriores cubriremos todos los ejemplos propuestos en el presente capítulo. No obstante, estos tercetos no recogen todas las posibilidades de ejecución básicas contempladas por un microprocesador actual. Por ejemplo, para llamar a una subrutina se tienen tercetos para meter los parámetros en la pila de llamadas, para invocar a la subrutina indicando el número de parámetros que se le pasa, para tomar un parámetro de la pila, y para retornar:

- **param x**: mete al parámetro real **x** en la pila.
- **call p, n**: llama a la subrutina que comienza en la etiqueta **p**, y le dice que tome **n** parámetros de la cima de la pila.
- **pop x**: toma un parámetro de la pila y lo almacena en la dirección **x**.
- **return y**: retorna el valor **y**.

Otros tercetos permiten gestionar el direccionamiento indirecto y el indexado:

- Direccionamiento indexado: los tercetos son de la forma **y := x[i]** y **x[i] := y**, donde **x** es la dirección base, e **i** es el desplazamiento.
- Direccionamiento indirecto: los tercetos son de la forma **y:=&x** y **x:=*y**, donde el símbolo **&** quiere decir "la dirección de ..." y el símbolo ***** quiere decir "el valor de la dirección contenida en la dirección ...".

La elección de operadores permisibles es un aspecto importante en el diseño de código intermedio. El conjunto de operadores debe ser lo bastante rico como para permitir implementar todas las operaciones del lenguaje fuente. Un conjunto de operadores pequeño es más fácil de implantar en una nueva máquina objeto, pero si es demasiado limitado puede obligar a la etapa inicial a generar largas secuencias de instrucciones para algunas operaciones del lenguaje fuente. En tal caso, el optimizador y el generador de código tendrán que trabajar más si se desea producir un buen código.

A continuación se muestra un ejemplo de código de tercetos que almacena en la variable **c** la suma de **a** y **b** (nótese que **b** se destruye como consecuencia de este cálculo):

```
        c = a
label etqBucle
        if b = 0 goto etqFin
        b = b-1
        c = c+1
        goto etqBucle
label etqFin
```

8.3 Una calculadora simple compilada

Para ilustrar como se utiliza el código de tercetos en una gramática vamos a suponer que nuestra calculadora en lugar de ser una calculadora interpretada es una calculadora compilada, es decir, en vez de interpretar las expresiones vamos a generar código intermedio equivalente.

8.3.1 Pasos de construcción

Antes de comenzar a codificar los analizadores léxico y sintáctico es necesario plantear exactamente qué se desea hacer y con qué gramática. Para ello se debe proponer un ejemplo preliminar de lo que debe hacer la calculadora para, a continuación, crear la gramática que reconozca el lenguaje, asignar los atributos necesarios y, una vez claro el cometido de las acciones semánticas, comenzar la codificación.

8.3.1.1 Propuesta de un ejemplo

El objetivo es que la calculadora produzca un texto de salida ante un texto de entrada. Por ejemplo, si la entrada es:

```
a := 5*b;
c := b := d := a*(4+v);
c := c := c;
```

la salida debería ser:

```
tmp1=5*b
a=tmp1
tmp2=4+v
tmp3=a*tmp2
d=tmp3
b=d
c=b
c=c
c=c
```

que es el código de tercetos equivalente.

8.3.1.2 Gramática de partida

La gramática de partida basada en reglas de producción es:

```
prog    :   asig ';'
        |   prog asig ';'
        |   error ';'
        |   prog error ';'
        ;
asig    :   ID ASIG expr
        |   ID ASIG asig
        ;
expr    :   expr '+' expr
        |   expr '*' expr
        |   '(' expr ')'
        |   ID
        |   NUMERO
        ;
```

Como puede observarse, no se permite el programa vacío, lo que hace que sea necesario introducir una nueva regla de error que gestione la posibilidad de que se produzca algún fallo en la primera asignación. Nótese que si se produce un error en esta

primera asignación, la pila del análisis sintáctico aún no tiene el no terminal **prog** a su izquierda, por lo que no es posible reducir por la regla **prog : prog error ';'**, pero sí por la regla **prog : error ';'**.

Por otro lado, la finalización de cada sentencia se hace con un punto y coma ';', en lugar de con retorno de carro, lo que resulta más cercano al tratamiento real de los lenguajes de programación actuales.

Además de permitirse las asignaciones múltiples, la última diferencia con respecto a la gramática del apartado 7.3.1 radica en que se ha hecho caso omiso de los tipos y, por tanto, tampoco tienen sentido las funciones de conversión de un tipo a otro.

Por último, esta gramática expresada en notación BNF es:

```
gramatica():{}{
        (sentFinalizada())*
}
sentFinalizada():{}{
            asigCompuesta() <PUNTOYCOMA>
        |   /* Gestión del error */
}
asigCompuesta():{}{
            LOOKAHEAD(4)
            <ID> ":=" asigCompuesta()
        |   <ID> ":=" expr()
}
expr():{}{
        term() ( "+" term() )*
}
term():{}{
        fact() ( "*" fact() )*
}
fact():{}{
            <ID>
        |   <NUMERO>
        |   "(" expr() ")"
}
```

8.3.1.3 Consideraciones importantes sobre el traductor

Para facilitar la lectura del código intermedio resultante, se hará uso de identificadores de variables en lugar de utilizar directamente sus direcciones de memoria. En cualquier caso, no hay que olvidar que la tabla de símbolos debe contener, junto con el identificador de cada variable, la dirección de memoria de cada una de ellas, con objeto de realizar la traducción correspondiente en el momento de generar el código máquina final.

La gramática que se va a utilizar será prácticamente la misma que la empleada en apartados anteriores. Recordemos que en cada regla en la que aparece un operador aritmético intervienen sólo dos expresiones, lo que encaja con la generación de un

terceto de asignación binaria, en el que sólo hay dos operandos y un resultado. Sin embargo, la reducción por cada una de estas reglas representa un resultado intermedio en el camino de calcular el resultado final, tal y como ilustra la figura 8.2.

Estos resultados intermedios deben almacenarse en algún lugar para que los tercetos puedan trabajar con ellos. Para este propósito se utilizan variables temporales generadas automáticamente por el compilador en la cantidad que sea necesario. La figura 8.3 muestra el código de tercetos que se debe generar para una asignación que tiene como r-valor la expresión de la figura 8.2, así como las variables temporales que es necesario generar en cada paso. Como resulta evidente, el código de tercetos resultante es equivalente al código fuente.

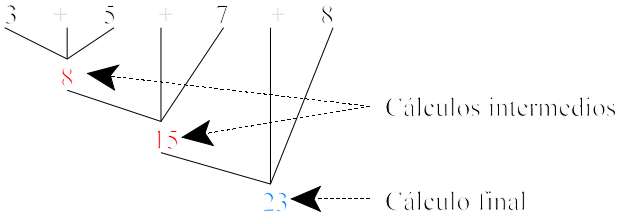


Figure 3 Los resultados intermedios se producen cada vez que se reduce por una regla de producción, e intervienen decisivamente en el cálculo del resultado final

Mediante el mecanismo de las variables temporales es posible construir, paso a paso, cualquier cálculo aritmético complejo, en el que pueden intervenir paréntesis,

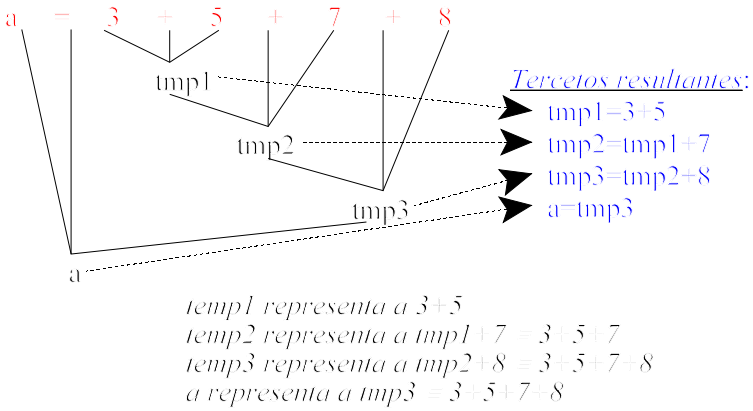


Figure 2 Representación de la variable temporal que hay que generar en cada reducción, y del terceto que debe producirse. El objetivo final de estos tercetos es almacenar en la variable **a** el valor **3+5+7+8**, lo que se consigue en base a las variables temporales

otras variables, etc. También resulta interesante observar que las variables temporales pueden reutilizarse en cálculos diferentes. No obstante, esto supone una optimización que no implementaremos en aras de concentrarnos exclusivamente en la generación de código correcto.

Para la realización de nuestra calculadora, hay que observar dos aspectos muy importantes:

- Gestionar adecuadamente los atributos. En la calculadora no hay parte de declaraciones ni chequeo de tipos, ya que se supone que todas las variables son de tipo numérico y que son declaradas en el momento en que hacen su primera aparición. Es por ello que no será necesaria una tabla de símbolos ya que, además, en los tercetos aparecerán los identificadores de las variables, y no las direcciones de memoria a que representan. Esto también nos permite omitir un gestor de direcciones de memoria que las vaya asignando a medida que van apareciendo nuevas variables.
- Realizar una secuencia de **printf**. Nuestro propósito es hacer una traducción textual, en la que entra un texto que representa asignaciones de alto nivel, y sale otro texto que representa a los tercetos. Por tanto nuestras acciones semánticas tendrán por objetivo realizar una secuencia de printf o System.out.println para visualizar estos textos de salida. Por este motivo, en tiempo de compilación, no se van a evaluar las expresiones ni a realizar cálculos de ningún tipo. Nuestro traductor se encarga únicamente de gestionar cadenas de texto; esto conlleva que, por ejemplo, el atributo asociado al token NUM no tenga porqué ser de tipo int, sino que basta con que sea un char[] o un String, ya que no va a ser manipulado aritméticamente.

En la construcción de un compilador real, la salida no suele ser un texto, sino un fichero binario que contiene el código máquina agrupado por bloques que serán gestionados por un enlazador (*linker*) para obtener el fichero ejecutable final. Asimismo, las acciones semánticas deben realizar toda la gestión de tipos necesaria antes de proceder a la generación de código, lo que lleva a la necesidad de una tabla de símbolos completa.

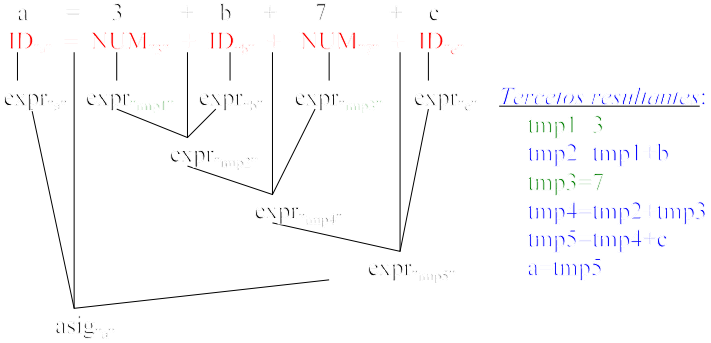
8.3.1.4 Atributos necesarios

Como ya se ha comentado, toda la gestión del código de entrada va a ser simbólica, en el sentido de que sólo se va a trabajar con cadenas de texto. De esta forma, tanto el terminal **ID** como el terminal **NUM** tendrán como atributo una cadena que almacena el lexema a que representan.

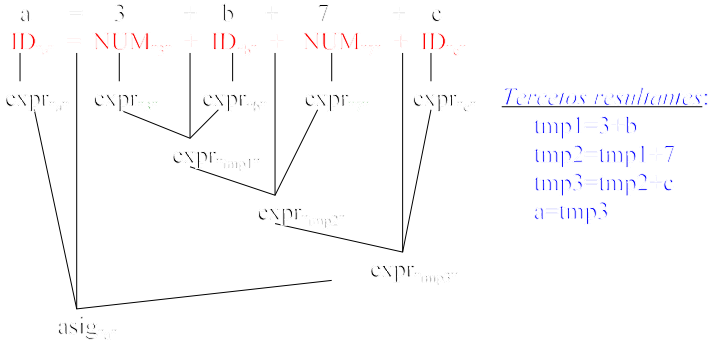
En cuanto a las expresiones, resulta sensato pensar que también deben tener como atributo a una cadena de caracteres que, de alguna forma, represente el trozo de árbol sintáctico del cual es raíz esa expresión. Retomando, por ejemplo, el árbol de la figura 8.3, se puede pensar en asignar a una expresión el texto que representa a cada variable, ya sea temporal o no; si asociamos el mismo tipo de atributo al no terminal

asig, entonces también se podrá generar código de tercetos incluso para las asignaciones múltiples. En cuanto a las expresiones que representan a un único valor numérico, se puede optar por generar una variable temporal para almacenarlos, o bien guardar el lexema numérico como atributo de la expresión, lo que conduce a una optimización en cuanto al número de tercetos generados y de variables temporales utilizadas. La figura 8.4.a) muestra el caso estricto en que sólo se permite como atributo el nombre de una variable, y la figura 8.4.b) muestra el caso relajado. En ambas situaciones el tipo de atributo es el mismo. En la solución que se propone más adelante se opta por el caso b), ya que supone una optimización sin ningún esfuerzo por parte del desarrollador.

Para acabar, la figura 8.5 muestra la propagación de atributos en el caso de realizar una asignación múltiple. Como ya se ha comentado antes, la generación de



Caso a). Una expresión representa siempre a una variable



Caso b). Una expresión representa a una variable o a un número

Figure 4 Asignación de atributos a los terminales y no terminales de la gramática de la calculadora. Em ambos casos el tipo de los atributos es el mismo. La diferencia entre el caso a) y el b) estriba en si se admite o no que una expresión pueda representar directamente a un literal numérico

código se hace posible mediante la utilización de un atributo al no terminal **asig** que representa al l-valor, en previsión de que sea, a su vez, el r-valor de una asignación

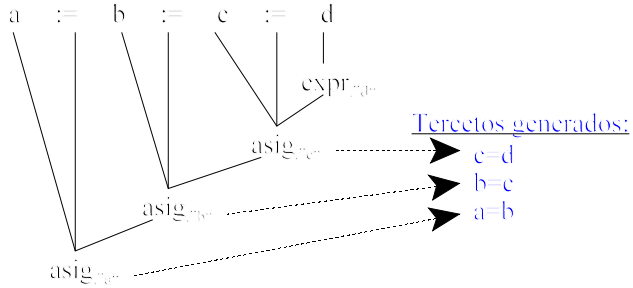


Figure 5 Asignaciones múltiples y el código generado equivalente

múltiple que se reduce de derecha a izquierda.

8.3.1.5 Acciones semánticas

Las acciones semánticas son sumamente sencillas, al no existir ninguna tabla de símbolos, ni haber chequeo de tipos. Tan sólo deben gestionarse convenientemente los atributos, propagándolos cuando sea necesario, generar las variables temporales y realizar los **printf** necesarios.

8.3.2 Solución con Lex/Yacc

En esta solución, el programa Lex se limita a reconocer los lexemas necesarios y a asignar a los *tokens* **NUM** e **ID** los lexemas que representan. El retorno de carro se considera un separador más. El código es:

```
%%
[0-9]+    {
                strcpy(yylval.cadena, yytext);
                return NUMERO;
          }

":="      { return ASIG; }
[a-zA-Z][a-zA-Z0-9]*   {
                                strcpy(yylval.cadena, yytext);
                                return ID;
                        }
[ \t\n]+  {;}
.         { return yytext[0]; }
```

El código Yacc también es muy escueto. Básicamente, cada reducción a una expresión o a una asignación, se encarga de generar un terceto, haciendo uso de variables temporales, si son necesarias. Éstas se crean mediante la función **nuevaTmp**.

Por otro lado, dado que todos los atributos son de tipo cadena de caracteres

podría pensarse en hacer uso directo de **YYSTYPE** en lugar de **%union**. Sin embargo, la herramienta PCLex no permite asignar a YYSTYPE punteros estáticos (un array de caracteres es, en C, un puntero estático), por lo que es necesario recurrir a la utilización del **%union** aunque ésta contenga un solo campo. El código resultante es:

```
%{
#include "stdio.h"
void nuevaTmp(char * s){
        static int actual=1;
        sprintf(s, "tmp%d", actual++);
}
%}
%union{
        char cadena[50];
}
%token <cadena> NUMERO ID
%token ASIG
%type <cadena> asig expr
%start prog
%left '+'
%left '*'
%%
prog    :   asig ';'
        |   prog asig ';'
        |   error ';'           { yyerrok; }
        |   prog error ';'      { yyerrok; }
        ;
asig    :   ID ASIG expr        {
                                  printf("%s=%s\n", $1, $3);
                                  strcpy($$, $1);
                                }
        |   ID ASIG asig        {
                                  printf("%s=%s\n", $1, $3);
                                  strcpy($$, $1);
                                }
        ;
expr    :   expr '+' expr {
                            nuevaTmp($$);
                            printf("%s=%s+%s\n", $$, $1, $3);
                          }
        |   expr '*' expr  {
                            nuevaTmp($$);
                            printf("%s=%s*%s\n", $$, $1, $3);
                          }
        |   '(' expr ')'    { strcpy($$, $2); }
        |   ID              { strcpy($$, $1); }
        |   NUMERO          { strcpy($$, $1); }
        ;
%%
#include "Calcul.c"
```

```
46 void main(){
47         yyparse();
48 }
49 void yyerror(char * s){
50         printf("%s\n",s);
51 }
```

8.3.3 Solución con JFlex/Cup

La solución con JFlex y Cup es casi idéntica a la ya vista en Lex/Yacc. Quizás la única diferencia radica en la función **nuevaTmp**, que hace uso de la variable estática **actual** inicializada a 1 en lugar de a 0 para facilitar la concatenación con el literal "tmp".

Así pues, el código en JFlex es:

```
 1 import java_cup.runtime.*;
 2 import java.io.*;
 3 %%
 4 %unicode
 5 %cup
 6 %line
 7 %column
 8 %%
 9 "+"        { return new Symbol(sym.MAS); }
10 "*"        { return new Symbol(sym.POR); }
11 "("        { return new Symbol(sym.LPAREN); }
12 ")"        { return new Symbol(sym.RPAREN); }
13 ";"        { return new Symbol(sym.PUNTOYCOMA); }
14 ":="       { return new Symbol(sym.ASIG); }
15 [:jletter:][:jletterdigit:]*        { return new Symbol(sym.ID, yytext()); }
16 [:digit:]+              { return new Symbol(sym.NUMERO, yytext()); }
17 [ \t\r\n]+                      {;}
18 .                       { System.out.println("Error léxico."+yytext()+"-"); }
```

Por su parte, el código Cup queda:

```
 1 import java_cup.runtime.*;
 2 import java.io.*;
 3 parser code {:
 4         public static void main(String[] arg){
 5             Yylex miAnalizadorLexico =
 6                     new Yylex(new InputStreamReader(System.in));
 7             parser parserObj = new parser(miAnalizadorLexico);
 8              try{
 9                 parserObj.parse();
10             }catch(Exception x){
11                 x.printStackTrace();
12                 System.out.println("Error fatal.\n");
13             }
14         }
15 :};
```

```
action code {:
        private static int actual=0;
        private static String nuevaTmp(){
            return "tmp"+(++actual);
        }
:}
terminal PUNTOYCOMA, MAS, POR;
terminal ASIG, LPAREN, RPAREN;
terminal String ID, NUMERO;
non terminal String asig, expr;
non terminal prog;
precedence left MAS;
precedence left POR;
/* Gramática */
prog    ::=  asig PUNTOYCOMA
        |    prog asig PUNTOYCOMA
        |    error PUNTOYCOMA
        |    prog error PUNTOYCOMA
        ;
asig    ::=  ID:i ASIG expr:e  {:
                                  System.out.println(i+"="+e);
                                  RESULT=i;
                                :}
        |    ID:i ASIG asig:a  {:
                                  System.out.println(i+"="+a);
                                  RESULT=i;
                                :}
        ;
expr    ::=  expr:e1 MAS expr:e2 {:
                                     RESULT=nuevaTmp();
                                     System.out.println(RESULT+"="+e1+"+"+e2);
                                   :}
        |    expr:e1 POR expr:e2 {:
                                     RESULT=nuevaTmp();
                                     System.out.println(RESULT+"="+e1+"*"+e2);
                                   :}
        |    LPAREN expr:e RPAREN {: RESULT=e; :}
        |    ID:i                  {: RESULT=i; :}
        |    NUMERO:n              {: RESULT=n; :}
        ;
```

8.3.4 Solución con JavaCC

La solución con JavaCC sigue exactamente las mismas pautas que las soluciones ascendentes. Quizás la única diferencia radica en que el código está más estructurado, al haberse ubicado en funciones independientes cada una de las acciones semánticas lo que, entre otras cosas, permite dar un tratamiento uniforme a las operaciones aritméticas mediante la función **usarOpAritmetico()**. Por otro lado, el reconocimiento de asignaciones múltiples se ha hecho mediante una regla recursiva por

la derecha y utilizando el LOOKAHEAD necesario (4 en este caso).

El código completo es:

```
1  PARSER_BEGIN(Calculadora)
2      import java.util.*;
3      public class Calculadora{
4          private static int actual=0;
5          public static void main(String args[]) throws ParseException {
6              new Calculadora(System.in).gramatica();
7          }
8          private static void usarASIG(String s, String e){
9              System.out.println(s+"="+e);
10         }
11         private static String usarOpAritmetico(String e1, String e2, String op){
12             actual++;
13             String tmp="tmp"+actual;
14             System.out.println(tmp+"="+e1+op+e2);
15             return tmp;
16         }
17     }
18 PARSER_END(Calculadora)
19 SKIP : {
20         " "
21     |   "\t"
22     |   "\r"
23     |   "\n"
24 }
25 TOKEN [IGNORE_CASE] :
26 {
27         <ID: ["A"-"Z"](["A"-"Z", "0"-"9"])*>
28     |   <NUMERO: (["0"-"9"])+>
29     |   <PUNTOYCOMA: ";">
30 }
31 /*
32 gramatica ::= ( sentFinalizada )*
33 */
34 void gramatica():{}{
35     (sentFinalizada())*
36 }
37 /*
38 sentFinalizada ::= ( ID ASIG )+ expr ';' | error ';'
39 */
40 void sentFinalizada():{}{
41     try {
42         asigCompuesta() <PUNTOYCOMA>
43     }catch(ParseException x){
44         System.out.println(x.toString());
45         Token t;
46         do {
```

```
47                  t = getNextToken();
48              } while (t.kind != PUNTOYCOMA);
49          }
50  }
51  String asigCompuesta():{
52          String s;
53          String a, e;
54  }{      LOOKAHEAD(4)
55              s=id() ":=" a=asigCompuesta()  { usarASIG(s, a); return s; }
56          |   s=id() ":=" e=expr()            { usarASIG(s, e); return s; }
57  }
58  /*
59  expr ::= term ('+' term)*
60  */
61  String expr():{
62          String t1, t2;
63  }{
64          t1=term() ( "+" t2=term() { t1=usarOpAritmetico(t1, t2, "+"); } )*       { return t1; }
65  }
66  /*
67  term ::= fact ('*' fact)*
68  */
69  String term():{
70          String f1, f2;
71  }{
72          f1=fact() ( "*" f2=fact() { f1=usarOpAritmetico(f1, f2, "*"); } )*       { return f1; }
73  }
74  /*
75  fact ::= ID | NUMERO | '(' expr ')'
76  */
77  String fact():{
78          String s, e;
79  }{
80              s=id()              { return s; }
81          |   s=numero()          { return s; }
82          |   "(" e=expr() ")"    { return e; }
83  }
84  String id():{}{
85          <ID>            { return token.image; }
86  }
87  String numero():{}{
88          <NUMERO> { return token.image; }
89  }
90  SKIP : {
91          <ILEGAL: (~[])>   { System.out.println("Carácter: "+image+" no esperado.");}
92  }
```

Aunque los no terminales **sentFinalizada()** y **asigCompuesta()** podrían haberse fusionado en uno sólo, se ha preferido diferenciar claramente entre la gestión

del error a través del terminal <**PUNTOYCOMA**>, y el reconocimiento de una asignación compuesta con recursión a derecha. Nótese asimismo que el no terminal **asigCompuesta()** engloba también el concepto de asignación simple como caso base de su recursión

8.4 Generación de código en sentencias de control

Una vez estudiado el mecanismo general de generación de código en expresiones aritméticas, el siguiente paso consiste en abordar el estudio del código de tercetos equivalente a las sentencias de alto nivel para el control del flujo de ejecución. Entre este tipo de sentencias están los bucles WHILE y REPEAT, así como las sentencias condicionales IF y CASE.

Como vimos en el apartado 8.2, el control del flujo de ejecución se realiza a bajo nivel mediante los tercetos **goto** y **call**. En este texto no entraremos en detalle sobre la gestión de subrutinas, por lo que nos centraremos exclusivamente en el **goto** incondicional y condicional.

8.4.1 Gramática de partida

Nuestro propósito consiste en reconocer programas que permitan realizar asignaciones a variables de tipo entero, y realizar cambios de flujo en base a la evaluación de condiciones. Las condiciones pueden ser simples o complejas; en las simples sólo aparecen operadores relacionales: >, <, >=, etc.; en las complejas se pueden utilizar los operadores lógicos **AND**, **OR** y **NOT** con el objetivo de concatenar condiciones más básicas.

Como puede observarse, se permiten bucles de tipo **WHILE** y de tipo **REPEAT**. En los primeros se evalúa una condición **antes** de entrar al cuerpo del bucle; si ésta es cierta se ejecuta el cuerpo y se comienza de nuevo el proceso; en caso contrario se continúa por la siguiente sentencia tras el **WHILE**. Por tanto el cuerpo se ejecuta **mientras** la condición que acompaña al **WHILE** sea **cierta**. Por otro lado, el bucle **REPEAT** evalúa la condición **tras** la ejecución del cuerpo, de forma que éste se ejecuta **hasta** que la condición del bucle sea **cierta**.

Como sentencias condicionales se permite el uso de la cláusula **IF** que permite ejecutar un bloque de código de forma opcional, siempre y cuando la condición sea **cierta**. Asimismo, un **IF** puede ir acompañado de una parte **ELSE** opcional que se ejecutaría caso de que la condición fuese falsa. Por otro lado, también se ha incluido la sentencia **CASE** que evalúa una expresión aritmética y la compara con cada uno de los elementos de una lista de valores predeterminada; caso de coincidir con alguno de ellos pasa a ejecutar el bloque de código asociado a dicho valor. Si no coincide con ninguno de ellos, pasa a ejecutar (opcionalmente) el bloque de código indicado por la cláusula **OTHERWISE**. Por último, se ha decidido no incluir la sentencia **FOR** con objeto de que el lector pruebe a codificar su propia solución, ya que resulta un ejercicio interesante.

Para ilustrar los mecanismos puros de generación de código, y no entremezclar en las acciones semánticas ninguna acción relativa a la gestión de tablas de símbolos ni control de tipos, se ha decidido que nuestro pequeño lenguaje sólo posea el tipo entero y que no sea necesario declarar las variables antes de su utilización.

La gramática que se va a utilizar, en formato de reglas de producción, es:

```
prog        :       prog sent ';'
            |       prog error ';'
            |       /* Épsilon */
            ;
sent        :       ID ASIG expr
            |       IF cond THEN sent ';' opcional FIN IF
            |       '{' lista_sent '}'
            |       WHILE cond DO sent ';' FIN WHILE
            |       REPEAT sent ';' UNTIL cond
            |       sent_case
            ;
opcional            :       ELSE sent';'
                    |       /*Epsilon*/
                    ;
lista_sent          :       /*Epsilon*/
                    |       lista_sent sent ';'
                    |       lista_sent error ';'
                    ;
sent_case           :       inicio_case OTHERWISE sent ';' FIN CASE
                    |       inicio_case FIN CASE
                    ;
inicio_case         :       CASE expr OF
                    |       inicio_case CASO expr ':'   sent ';'
                    ;
expr        :       NUMERO
            |       ID
            |       expr'+'expr
            |       expr'-'expr
            |       expr'*'expr
            |       expr'/'expr
            |       '-'expr %prec MENOS_UNARIO
            |       '('expr')'
            ;
cond        :       expr'>'expr
            |       expr'<'expr
            |       expr'MAI'expr
            |       expr'MEI'expr
            |       expr'='expr
            |       expr'DIF'expr
            |       NOT cond
            |       cond AND cond
            |       cond OR cond
            |       '(' cond ')'
```

;

Es de notar que el cuerpo de una sentencia **WHILE**, **REPEAT**, **IF**, etc. está formado por una sola sentencia. Si se desea incluir más de una, el mecanismo consiste en recurrir a la sentencia compuesta. Una sentencia compuesta está formada por una lista de sentencias delimitada por llaves, tal y como indica la regla:

```
sent :    '{' lista_sent '}'
```

Además, nada impide introducir como cuerpo de cualquier sentencia de control de flujo a otra sentencia de control de flujo. Es más, el anidamiento de sentencias puede producirse a cualquier nivel de profundidad. Este hecho hace que haya que prestar especial atención al control de errores. Como puede observarse en la gramática, se tienen dos reglas de error, una a nivel de programa y otra a nivel de **lista_sent**, esto es, en el cuerpo de una sentencia compuesta.

Si se produce un error sintáctico en el interior de una sentencia compuesta, y sólo se tuviera la regla de error a nivel de programa, se haría una recuperación incorrecta, ya que el mecanismo *panic mode* extraería elementos de la pila hasta encontrar **prog**; luego extraería *tokens* hasta encontrar el punto y coma. Y por último continuaría el proceso de análisis sintáctico pensando que ha recuperado el error convenientemente y que las siguientes sentencias están a nivel de programa, lo cual es falso puesto que se estaba en el interior de una sentencia compuesta. Este problema se ilustra en la figura 8.6. Para evitarlo es necesario introducir una nueva regla de error a nivel de sentencia compuesta.

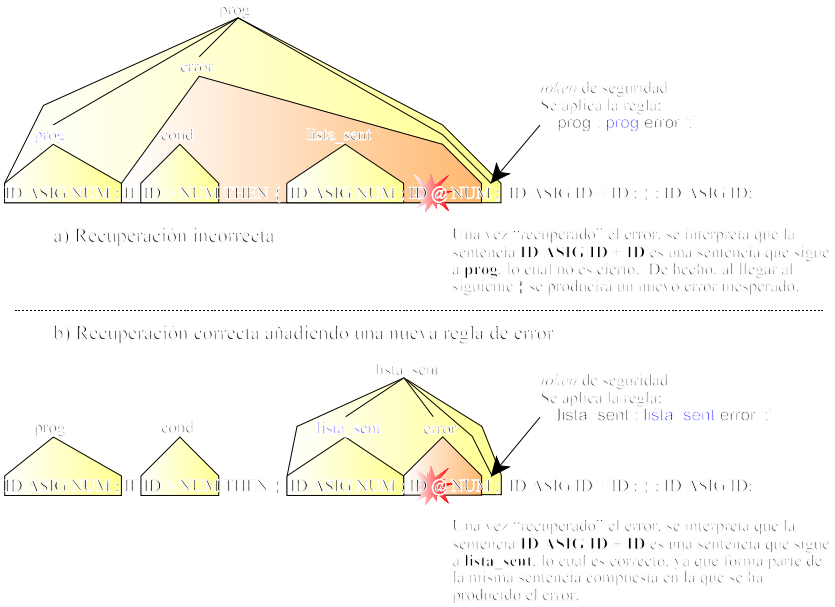


Figure 6 Necesidad de dos reglas de error cuando se utilizan sentencias compuestas

Desde el punto de vista BNF para JavaCC, la gramática queda:

```
void gramatica():{}{
        (sentFinalizada())*
}
void sentFinalizada():{}{
        (
                s=id() <ASIG> e=expr()
        |       <IF> cond() <THEN> sentFinalizada()
                [<ELSE> sentFinalizada()]
                <FIN> <IF>
        |       <WHILE> cond() <DO>
                        sentFinalizada()
                <FIN> <WHILE>
        |       <REPEAT>
                        sentFinalizada()
                <UNTIL> cond()
        |       <LLLAVE> gramatica() <RLLAVE>
        |       <CASE> expr() <OF>
                        (<CASO> expr() <DOSPUNTOS> sentFinalizada() )*
                        [<OTHERWISE> sentFinalizada()]
                <FIN> <CASE>
        ) <PUNTOYCOMA>
```

```
        | /* Gestión del error */
}
String expr():{}{
        term()    ( <MAS  | <MENOS> term() )*
}
String term():{}{
        fact()    ( <POR> | <ENTRE> fact() )*
}
String fact():{}{
        (<MENOS>)*
        (
                <ID>
        |       <NUMERO>
        |       <LPAREN> expr() <RPAREN>
        )
}
BloqueCondicion cond():{}{
        condTerm()        ( <OR> condTerm() )*
}
BloqueCondicion condTerm():{}{
        c1=condFact()     ( <AND> condFact() )*
}
BloqueCondicion condFact():{}{
        (<NOT> )*
        (       condSimple()
        |       <LCOR> cond() <RCOR>
        )
}
BloqueCondicion condSimple():{}{
        expr()  (                       <MAYOR> expr()
                                |       <MENOR> expr()
                                |       <IGUAL> expr()
                                |       <MAI> expr()
                                |       <MEI> expr()
                                |       <DIF> expr()
                                )
}
```

Esta gramática tiene dos diferencias con respecto a la expresada con reglas de producción. En primer lugar el no terminal **lista_sent** ha desaparecido, ya que sintácticamente es equivalente a **prog** (que aquí hemos llamado **gramática**). Ello nos lleva a necesitar un solo control de errores en la última regla del no terminal **sentFinalizada**. Evidentemente, esto también podría haberse hecho en la gramática basada en reglas, pero se ha preferido ilustrar la gestión de varias reglas de error en una misma gramática.

La segunda diferencia radica en la gestión de las condiciones mediante una gramática descendente. Ante una secuencia de caracteres como: "((((27*8) ..." es necesario conocer el contenido de los puntos suspensivos para poder decidir si los tres

primeros paréntesis se asocian a una expresión o a una condición: si nos encontramos el carácter “>” los paréntesis se asocian a una condición, pero si nos encontramos “-7)” entonces, al menos, el tercer paréntesis es de una expresión y aún habría que seguir mirando para ver a qué pertenecen los otros dos paréntesis. Este problema tiene tres soluciones claramente diferenciadas:

- Establecer un *lookahead* infinito.
- Diferenciar las agrupaciones de expresiones y las de condiciones; por ejemplo, las expresiones se pueden agrupar entre paréntesis, y las condiciones entre corchetes. Esta ha sido la solución adoptada en este ejemplo.
- Considerar que una condición es una expresión de tipo lógico o *booleano*. Esta es la decisión más ampliamente extendida, aunque no se ha elegido en este caso ya que implica una gestión de tipos sobre la que no nos queremos centrar.

8.4.2 Ejemplos preliminares

A continucación se ilustran unos cuantos ejemplos de programas válidos reconocidos por nuestra gramática, así como el código de tercetos equivalente que sería deseable generar.

El primer ejemplo ilustra el cálculo iterativo del factorial de un número **n**:

```
ALFA := n;
FACTORIAL := 1;
WHILE ALFA > 1 DO
 {
  FACTORIAL := FACTORIAL * ALFA;
  ALFA := ALFA - 1;
 };
FIN WHILE;
```

El código de tercetos equivalente sería:

```
ALFA = n
        tmp1 = 1;
        FACTORIAL = tmp1
label etq1
        tmp2 = 1;
        if ALFA > tmp2 goto etq2
        goto etq3
label etq2
        tmp3 = FACTORIAL * ALFA;
        FACTORIAL = tmp3
        tmp4 = 1;
        tmp5 = ALFA - tmp4;
        ALFA = tmp5
        goto etq1
label etq3
```

El siguiente ejemplo ilustra el código generado para una sentencia CASE:

```
CASE NOTA OF
 * Podemos emplear comentarios, dedicando una linea a cada uno de ellos.
 CASO 5 : CALIFICACION := SOBRESALIENTE;
 CASO 4 : CALIFICACION := NOTABLE;
 CASO 3 : CALIFICACION := APROBADO;
 CASO 2 : CALIFICACION := INSUFICIENTE;
OTHERWISE
 CALIFICACION := MUY_DEFICIENTE;
FIN CASE;
```

que equivale a:

```
tmp1 = 5;
        if NOTA != tmp1 goto etq2
        CALIFICACION = SOBRESALIENTE
        goto etq1
label etq2
        tmp2 = 4;
        if NOTA != tmp2 goto etq3
        CALIFICACION = NOTABLE
        goto etq1
label etq3
        tmp3 = 3;
        if NOTA != tmp3 goto etq4
        CALIFICACION = APROBADO
        goto etq1
label etq4
        tmp4 = 2;
        if NOTA != tmp4 goto etq5
        CALIFICACION = INSUFICIENTE
        goto etq1
label etq5
        CALIFICACION = MUY_DEFICIENTE
label etq1
```

Para finalizar, el siguiente ejemplo demuestra cómo es posible anidar sentencias compuestas a cualquier nivel de profundidad:

```
        JUGAR := DESEO_DEL_USUARIO;
        WHILE JUGAR = VERDAD DO
         {
          TOTAL := 64;
          SUMA_PUNTOS := 0;
          NUMERO_TIRADAS := 0;
          TIRADA_ANTERIOR := 0;
          REPEAT
           {
            DADO := RANDOMIZE * 5 + 1;
            IF TIRADA_ANTERIOR != 6 THEN
              NUMERO_TIRADAS := NUMERO_TIRADAS + 1;
            FIN IF;
```

```
        SUMA_PUNTOS := SUMA_PUNTOS + DADOS;
        IF SUMA_PUNTOS > TOTAL THEN
          SUMA_PUNTOS := TOTAL -(SUMA_PUNTOS - TOTAL);
        ELSE
          IF SUMA_PUNTOS != TOTAL THEN
            CASE DADO OF
              CASO 1: UNOS := UNOS + 1;
              CASO 2: DOSES := DOSES + 1;
              CASO 3: TRESES := TRESES + 1;
              CASO 4: CUATROS := CUATROS + 1;
              CASO 5: CINCOS := CINCOS + 1;
            OTHERWISE
                  SEISES := SEISES + 1;
            FIN CASE;
          FIN IF;
        FIN IF;
        TIRADA_ANTERIOR := DADO;
       };
       UNTIL SUMA_PUNTOS = TOTAL;
       JUGAR := DESEO_DEL_USUARIO;
     };
    FIN WHILE;
```

En este caso, el código generado es verdaderamente enrevesado:

```
      JUGAR = DESEO_DEL_USUARIO
label etq1
      if JUGAR = VERDAD goto etq2
      goto etq3
label etq2
      tmp1 = 64;
      TOTAL = tmp1
      tmp2 = 0;
      SUMA_PUNTOS = tmp2
      tmp3 = 0;
      NUMERO_TIRADAS = tmp3
      tmp4 = 0;
      TIRADA_ANTERIOR = tmp4
label etq4
      tmp5 = 5;
      tmp6 = RANDOMIZE * tmp5;
      tmp7 = 1;
      tmp8 = tmp6 + tmp7;
      DADO = tmp8
      tmp9 = 6;
      if TIRADA_ANTERIOR != tmp9 goto etq5
      goto etq6
label etq5
      tmp10 = 1;
      tmp11 = NUMERO_TIRADAS + tmp10;
      NUMERO_TIRADAS = tmp11
      gotoetq7
label etq6
label etq7
      tmp12 = SUMA_PUNTOS + DADOS;
      SUMA_PUNTOS = tmp12
      if SUMA_PUNTOS > TOTAL goto etq8
      goto etq9
label etq8
      tmp13 = SUMA_PUNTOS - TOTAL;
      tmp14 = TOTAL - tmp13;
      SUMA_PUNTOS = tmp14
      gotoetq10
label etq9
      if SUMA_PUNTOS != TOTAL goto etq11
      goto etq12
label etq11
      tmp15 = 1;
      if DADO != tmp15 goto etq14
      tmp16 = 1;
      tmp17 = UNOS + tmp16;
      UNOS = tmp17
      goto etq13
label etq14
      tmp18 = 2;
      if DADO != tmp18 goto etq15
      tmp19 = 1;
      tmp20 = DOSES + tmp19;
      DOSES = tmp20
      goto etq13
label etq15
```

```
        tmp21 = 3;
        if DADO != tmp21 goto etq16
        tmp22 = 1;
        tmp23 = TRESES + tmp22;
        TRESES = tmp23
        goto etq13
label etq16
        tmp24 = 4;
        if DADO != tmp24 goto etq17
        tmp25 = 1;
        tmp26 = CUATROS + tmp25;
        CUATROS = tmp26
        goto etq13
label etq17
        tmp27 = 5;
        if DADO != tmp27 goto etq18
        tmp28 = 1;
        tmp29 = CINCOS + tmp28;
        CINCOS = tmp29
        goto etq13
label etq18
        tmp30 = 1;
        tmp31 = SEISES + tmp30;
        SEISES = tmp31
label etq13
        goto etq19
label etq12
label etq19
label etq10
        TIRADA_ANTERIOR = DADO
        if SUMA_PUNTOS = TOTAL goto etq20
        goto etq21
label etq21
        goto etq4
label etq20
        JUGAR = DESEO_DEL_USUARIO
        goto etq1
label etq3
```

8.4.3 Gestión de condiciones

Todos los cambios de flujo que admite el lenguaje que acabamos de definir son condicionales. En otras palabras, en base a la veracidad o falsedad de una condición, el programa sigue su flujo de ejecución por una sentencia o por otra. Es por esto que el código que decidamos generar para evaluar una condición será decisivo para gestionar el flujo en las diferentes sentencias de control.

A este respecto, hay dos posibilidades fundamentales. Quizás la más sencilla estriba en considerar que las condiciones son expresiones especiales de tipo lógico y cuyo valor de evaluación es 0 ó 1 (falso ó verdadero respectivamente) y se almacena en una variable temporal al igual que ocurre con las expresiones de tipo entero. La sentencia en la que se enmarca esta condición preguntará por el valor de la variable temporal para decidir por dónde continúa el flujo. Por ejemplo, la regla:

sent : IF cond THEN sent opcional FIN IF

↑ ① ↑ ② ↑ ③

establecería en una acción ubicada en el punto ① un terceto del tipo

if tmpXX=0 goto etqFalso

donde **tmpXX** es la variable en la que se almacena el resultado de haber evaluado la condición. Si su valor es diferente de 0, o sea cuando la condición es verdad, entonces no se produce el salto y se continúa el flujo de ejecución, que se corresponde con la sentencia del cuerpo del IF. El resultado es que esta sentencia sólo se ejecuta cuando la condición es cierta. Como opcional se corresponde con el cuerpo del **ELSE** (posiblemente vacío), pues es aquí exactamente adonde se debe saltar en caso de que la condición sea falsa. En cualquier caso, una vez ejecutado el cuerpo del **IF** se debe saltar detrás del **FIN IF**, con objeto de no ejecutar en secuencia la parte opcional del **ELSE**. Por tanto, en el punto ② deben colocarse los tercetos:

goto etqFINIF

label etqFalso:
Finalmente la etiqueta de fin del **IF** debe ubicarse detrás de la sentencia completa, en el punto ③:
label etqFINIF:
con lo que el código completo generado sería:

```
        // Código que evalúa cond y mete su valor en tmpXX
        if tmpXX=0 goto etqFalso
        // Código de la sentencia del IF
        goto etgFINIF
label etqFalso:
        // Código de la sentencia del ELSE
label etqFINIF:
```

Para realizar este propósito, resulta fundamental la inserción de acciones semánticas intercaladas en el consecuente de una acción semántica. Además, las etiquetas que aquí se han identificado como etqFalso y etqFINIF deben ser, en realidad, etiquetas cualesquiera generadas ad hoc por nuestro compilador, por ejemplo etq125 y etq563 respectivamente.

En cualquier caso, este mecanismo de evaluación de condiciones incurre en una disminución de la eficiencia ya que las condiciones compuestas deben evaluarse al completo. Para evitar este problema sin enrarecer el código, se utiliza una técnica que permite la evaluación de condiciones usando cortocircuito. Por ejemplo, en una condición compuesta de la forma **cond$_1$ OR cond$_2$**, sólo se evaluará la segunda condición si la primera es falsa ya que, de lo contrario, se puede asegurar que el resultado del **OR** será verdadero independientemente del valor de cond$_2$. Una cosa parecida sucede con el **AND**. Éste será el método que emplearemos, y que se explica con detalle a continuación.

8.4.3.1 Evaluación de condiciones usando cortocircuito

El mecanismo consiste en que cada **cond** genere un bloque de código que posee dos saltos (**goto**): un salto a una etiqueta **A** caso de cumplirse la condición, y un salto a otra etiqueta **B** caso de no cumplirse la condición.

Generar este código para las condiciones simples resulta trivial. A partir de ellas se estudiará el cortocircuito en condiciones compuestas.

8.4.3.1.1 Condiciones simples

Como punto de partida se asume que las expresiones y las asignaciones generan el mismo código intermedio que se estudió en la calculadora del apartado 8.3. Por ello, ante una condición como por ejemplo:

```
7*a > 2*b-1
```

se utilizará la regla:

cond : expr$_1$ '>' expr$_2$

y por las reglas de **expr**, y sin introducir todavía ninguna acción semántica en la regla de **cond**, se genera el código:

❶ tmp1=7*a
❷ tmp2=2*b
❸ tmp3=tmp2-1

donde la línea ❶ se corresponde con $expr_1$ y las líneas ❷ y ❸ se corresponden con $expr_2$. Además, el atributo que representa a $expr_1$ tiene el valor "tmp1", mientras que quien representa a $expr_2$ es "tmp3".

Por otro lado, pretendemos que una regla de producción de una condición simple como:

cond : $expr_1$ op_{rel} $expr_2$

genere un bloque de código de la forma:

if var_1 op_{rel} var_2 goto etq_{Verdad}
goto etq_{Falso}

donde var_1 es el representante de la primera expresión, ya sea una variable temporal, una variable de usuario o una constante numérica; var_2 representa a la segunda expresión; y op_{rel} representa al operador de comparación: mayor que (>), menor que (<), distinto de (<>), etc. Además, etq_{Verdad} y etq_{Falso} son etiquetas generadas *ad hoc* por el compilador de manera secuencial: etq001, etq002, etc., de forma parecida a como se generaban las variables temporales. Siguiendo con nuestro ejemplo, el código que nos interesa producir es:

if tmp1 > tmp3 goto etq001
goto etq002

donde etq001 representa la etq_{Verdad} y etq002 representa etq_{Falso}. Para generar este código basta con asociar una acción semántica a la regla de la condición, de forma que ésta quedaría:

```
cond : expr1 '>' expr2    {
                              nuevaEtq($$.etqVerdad);
                              nuevaEtq($$.etqFalso);
                              printf("\tif %s > %s goto %s\n", $1, $3, $$.etqVerdad);
                              printf("\tgoto %s\n", $$.etqFalso);
                          }
```

Como puede observarse en esta acción semántica, los atributos asociados a una condición son un par de etiquetas, una de verdad y otra de falso. Además, el bloque de código g enerado posee dos **gotos** (uno condicional y otro incondicional), a sendas etiquetas, pero no ha ubicado el destino de ninguna de ellas, esto es, no se han puesto

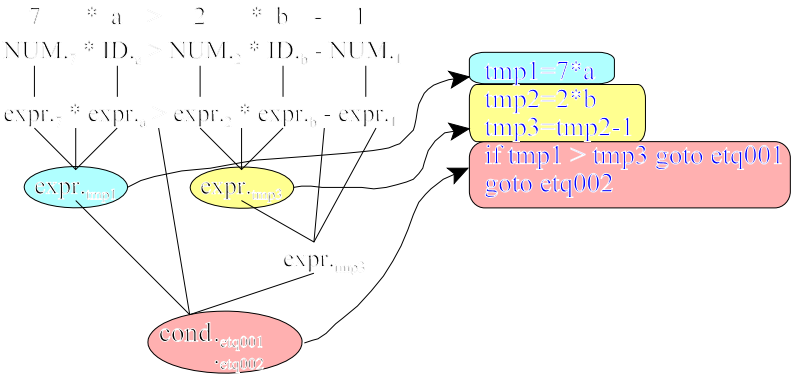


Figure 7 Generación de código para una condición simple

los **label**. La figura 8.8 muestra el código asociado a cada bloque gramatical.

Esto se debe a que la regla que haga uso de la condición (ya sea en un IF, en un WHILE, etc.), debe colocar estos **label** convenientemente. Por ejemplo, un WHILE colocaría el destino de la etiqueta de verdad al comienzo del cuerpo del bucle, mientras que la de falso iría tras el cuerpo del bucle; de esta manera cuando la condición sea falsa se salta el cuerpo, y cuando sea cierta se ejecuta un nuevo ciclo. Una cosa parecida ocurriría con el resto de las sentencias; la figura 8.7 muestra el caso de un **IF**.

8.4.3.1.2 Condiciones compuestas

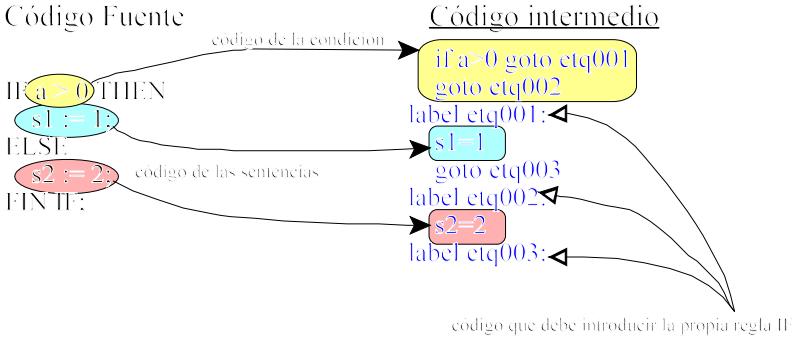


Figure 8Código que debe generar una sentencia IF completa

Un hecho que se deduce de la discusión anterior es que las sentencias en las que se enmarca una condición (IF, WHILE, etc.) deben realizar un tratamiento uniforme de éstas tanto si se trata de condiciones simples como de condiciones compuestas en las que intervienen los operadores lógicos AND, OR y NOT. Por tanto, una condición compuesta debe generar por sí misma un bloque de código en el que existan dos **gotos** para los cuales no se ha establecido aún el destino. Las etiquetas de destino se asignarán como atributos del no terminal **cond**, con lo que el tratamiento de las condiciones compuestas coincide con el de las simples.

Además, se desea utilizar la técnica del cortocircuito, por el que una condición sólo se evalúa si su valor de verdad o falsedad es decisivo en la evaluación de la condición compuesta en la que interviene.

La explicación siguiente puede comprenderse mejor si pensamos en el bloque de código generado por una condición como una caja negra que ejecuta, en algún momento, o un salto a una etiqueta de falso, o uno a una etiqueta de verdad, tal y como ilustra la figura 8.9.a). Por tanto, ante una regla como

cond : $cond_1$ AND $cond_2$

se generarían los bloques de código de la figura 8.9.b). Pero, al tener que reducir todo el conjunto a una condición compuesta, los bloques de la figura 8.9.b) deben reducirse a uno sólo, en el que exista una sola etiqueta de verdad y una sola de falso, tal y como ilustra la figura 8.9.c). Asumimos que el flujo le llega a un bloque por arriba.

En otras palabras, la figura 8.9.c) muestra que, sin asociar todavía ninguna acción semántica a la regla

cond : $cond_1$ AND $cond_2$

se generarían bloques de código que poseen cuatros **gotos** pero ningún **label**. Por tanto

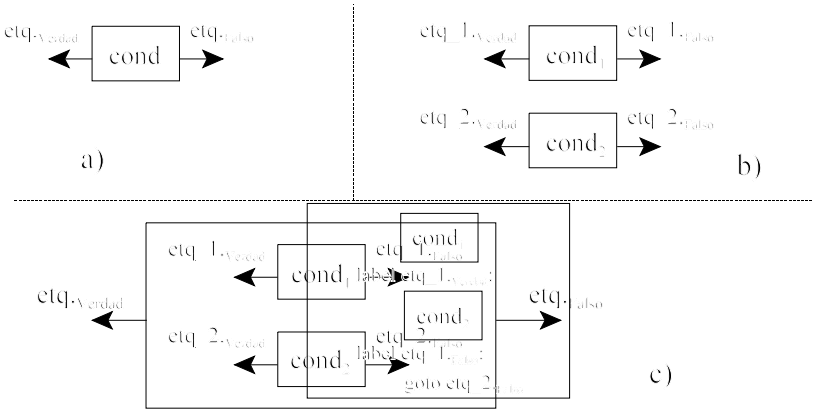


Figure 9 Diagramas de bloques asociados a las condiciones

Figure 10Código asociado a una condición compuesta por AND

nuestro objetivo es:

- Reducir los cuatro **gotos** sin **label** a tan sólo dos **gotos** y que coincidan con las etiquetas de verdad y falso del bloque cond general.
- Que la segunda condición sólo se evalúe cuando la primera es cierta; en caso contrario la técnica del cortocircuito no necesita evaluarla ya que la condición general es falsa.

De esta forma, el resultado general es que habremos generado un solo bloque de código grande con una sola etiqueta de verdad y una sola de falso. Al obtener un único gran bloque de código, desde el punto de vista externo podremos tratar una condición de manera uniforme, ya se trate de una condición simple o una compuesta.

Podemos conseguir estos dos objetivos de una manera muy sencilla; el método consiste en colocar los **label** de $cond_1$ estratégicamente, de forma que se salte al código que evalúa $cond_2$ sólo cuando se cumple $cond_1$; cuando $cond_1$ sea falsa, $cond_2$ no debe evaluarse, sino que debe saltarse al mismo sitio que si $cond_2$ fuera falsa, $etq\_2._{Falso}$. De esta manera, si alguna de las condiciones es falsa, el resultado final es que se salta a $etq\_2._{Falso}$, y si las dos son ciertas, entonces se saltará a $etq\_2._{Verdad}$. Por tanto, las etiquetas de $cond_2$ coincidirán con las etiquetas de la condición global, y habrá que colocar acciones semánticas para escribir los **label** tal y como se ha indicado. El diagrama de bloques quedaría tal y como se indica en la figura 8.10.

Si sustituimos las flechas por código de tercetos nos queda un bloque como el de la figura 8.11. Como puede observarse, para generar la etiqueta entre las dos

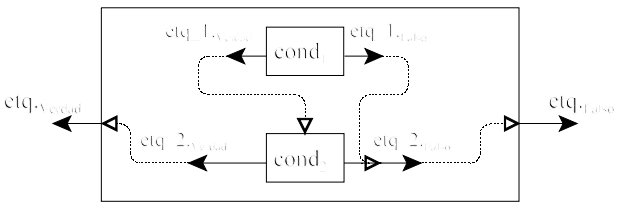


Figure 11 Diagrama de bloques asociado a una condición compuesta AND

condiciones será necesario incluir una acción semántica intermedia. Asimismo, también puede apreciarse en esta figura el subterfugio utilizado para conseguir que el destino de falsedad para ambas condiciones sea el mismo: basta con ubicar una sentencia de salto incondicional a la segunda etiqueta justo después del destino de la primera. Además, las etiquetas de verdad y de falso del bloque completo se corresponden con las de verdad y falso de $cond_2$, o sea $etq\_2._{Verdad}$ y $etq\_2._{Falso}$, por lo que ambos atributos deben copiarse tal cual desde $cond_2$ hasta el antecedente cond en la correspondiente regla de producción. La figura 8.12 muestra un ejemplo de código generado para el caso de la condición: a > 3 AND a < 8.

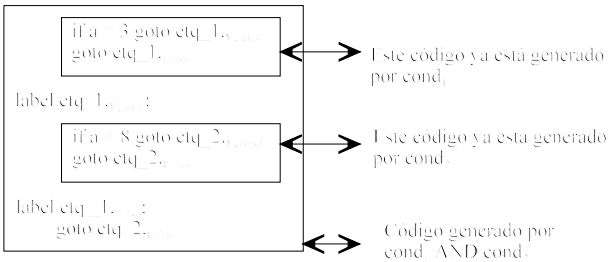


Figure 12Código generado para la condición: a>3 AND a<8

Las acciones semánticas asociadas a la regla de producción interviniente serían:

```
cond :   cond AND      {  printf("label %s\n", $1.etqVerdad);}
           cond        {  printf ("label %s\n", $1.etqFalso);
                          printf ("\tgoto %s\n", $4.etqFalso);
                          strcpy($$.etqVerdad, $4.etqVerdad);
                          strcpy($$.etqFalso, $4.etqFalso);
                       }
```

Para comprender bien estas acciones semánticas, debemos recordar que una acción intermedia es traducida por PCYacc a un nuevo no terminal y que, por tanto, cuenta a la hora de numerar los símbolos del consecuente; por ello los atributos asociados a la segunda condición son referidos como $4 y no como $3; $3 es, realmente, la propia acción intermedia. Por otro lado, la acción intermedia podría haberse colocado también justo antes del *token* **AND** habiéndose producido idénticos resultados. No obstante, cuando una acción semántica pueda colocarse en varios puntos de una regla, es conveniente retrasarla lo máximo posible pues con ello pueden evitarse conflictos desplazar/reducir en el analizador sintáctico.

El tratamiento dado a las condiciones compuestas ha sido sistemático y uniforme con respecto a las condiciones simples. Ello nos permite generar grandes bloques de código que, a su vez, pueden intervenir en otras condiciones compuestas. En otras palabras, las acciones semánticas propuestas permiten generar código correctamente para condiciones de la forma: $cond_1$ AND $cond_2$ AND $cond_3$ ... AND $cond_n$.

Como el lector ya ha podido vaticinar, el tratamiento del operador lógico OR obedece a un patrón similar al estudiado para el AND, con la salvedad de que se intercambian los papeles de las etiquetas de verdad y falso en la primera condición, lo que produce un diagrama de bloques como el de la figura 8.13.

La figura 8.14 muestra un ejemplo de código generado para la condición a=3 OR a=5, mientras que las acciones semánticas asociadas a la regla de producción interviniente serían:

```
cond :    cond OR        {  printf("label %s\n", $1.etqFalso);}
             cond        {  printf ("label %s\n", $1.etqVerdad);
                            printf ("\tgoto %s\n", $4.etqVerdad);
                            strcpy($$.etqVerdad, $4.etqVerdad);
                            strcpy($$.etqFalso, $4.etqFalso);
                }
```

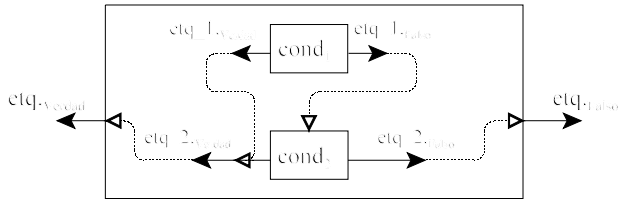


Figure 13 Diagrama de bloques asociado a una condición compuesta OR

Por último, el operador NOT tiene un tratamiento muy particular. Si se piensa bien, cuando se reduce por la regla

cond : NOT $cond_1$

el analizador sintáctico habrá ejecutado las acciones semánticas asociadas a la condición del consecuente ($cond_1$), por lo que ya sa habrá generado todo el código necesario para saber si ésta es cierta o no. Por tanto, no tiene sentido generar más código asociado a la aparición del operador NOT, sino que basta con intercambiar los papeles de las etiquetas de verdad y falso del consecuente, que pasarán a ser las etiquetas de falso y de verdad, respectivamente, del antecedente. En otras palabras, cuando la $cond_1$ es falsa, se produce un salto a la etiqueta de verdad del antecedente, ya que NOT $cond_1$ es cierta.

La figura 8.15 muestra el diagrama de bloques del procedimiento explicado. Por otro lado, la figura 8.16 muestra un ejemplo de los atributos asociados a la condición "a > 3" y a la condición "NOT a > 3", siendo el código generado el mismo

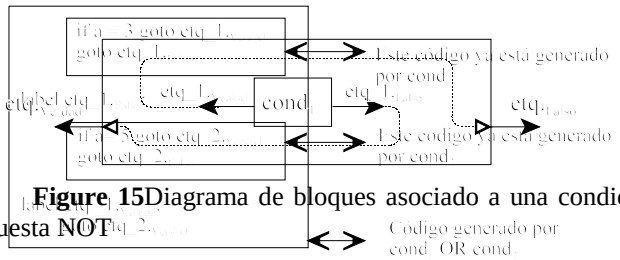


Figure 15 Diagrama de bloques asociado a una condición compuesta NOT

Figure 14Código generado para la condición: a=3 OR a=5

en ambos casos.

La acción semántica asociada a la regla de producción interviniente es:

```
cond : NOT cond {  strcpy($$.etqVerdad, $2.etqFalso);
                   strcpy ($$.etqFalso, $2.etqVerdad);
                }
```

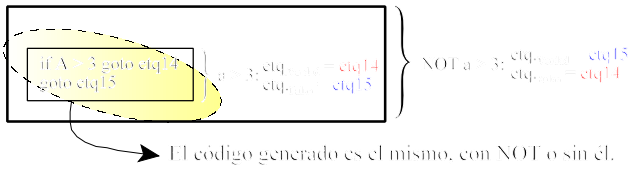


Figure 16Código generado para la condición: NOT a>3

Con esto finalizamos todos los operadores lógicos que posee el lenguaje propuesto. Para otros operadores, tales como NAND, NOR, IMPLIES (implicación lógica), etc., el procedimiento es el mismo. Un caso particular es el del operador XOR, cuya tabla de verdad es:

A	B	A XOR B
Verdad	Verdad	Falso
Verdad	Falso	Verdad
Falso	Verdad	Verdad
Falso	Falso	Falso

Este operador no admite cortocircuito por lo que se debe generar código apoyado por variables temporales para que se salte a una etiqueta de falso caso de que las dos condiciones sean iguales, y a una etiqueta de verdad en caso contrario.

8.4.4 Gestión de sentencias de control de flujo

En este punto abordaremos la utilización de las condiciones y sus etiquetas asociadas, para generar el código de sentencias que alteran el flujo secuencial de ejecución en función del valor de tales condiciones. Estas sentencias son: IF, WHILE, CASE y REPEAT.

8.4.4.1 Sentencia IF-THEN-ELSE

El caso de la sentencia **IF** es el más simple. Aquí basta con indicar que la etiqueta de verdad de la condición está asociada al código a continuación del **THEN**, y la etiqueta de falso se asocia al código que puede haber tras el **ELSE**. En cualquier caso, una vez acabadas las sentencias del **THEN** se debe producir un salto al final del **IF**, porque no queremos que se ejecuten también las sentencias del **ELSE**. Por tanto, tras la sentencias del **THEN**, creamos una nueva etiqueta a la cual produciremos un salto, y colocamos el destino de tal etiqueta al final del código del **IF**. Esto puede apreciarse mejor con un ejemplo:

Código fuente	Código intermedio

IF A > 0 THEN
  S1 := 1;
ELSE
  S2 := 2;
FIN IF
	if A>0 goto etq1
	goto etq2
	label etq1:
	S1 = 1
	goto etq3
	label etq2:
	S2 = 2
	label etq3:

El diagrama de bloques puede apreciarse en la figura 8.17.

Grosso modo, para hacer esto se necesita insertar dos acciones semánticas intermedias, una que visualice el código marcado en rojo y otra para el código marcado en azul. La etiqueta **etq3** se almacena en una variable a la que llamaremos etqFINIF; así, las acciones semánticas necesarias quedarían como se indica a continuación (en este sencilla a proximación se supone que todo **IF** tiene un **ELSE**; más adelante se propondrá el caso general):

{ printf("label %s\n", $2.etqVerdad); }
↓
sent : IF cond THEN sent ELSE sent FIN IF { printf("label %s:\n", *etqFINIF*); }
↑
{ nuevaEtq(*etqFINIF*);
printf ("\tgoto %s\n", *etqFINIF*);
printf ("label %s:\n", $2.etqFalso); }

Sin embargo, aquí aparece un problema que se convertirá en una constante en el resto de sentencias de control: ¿dónde se declara la variable etqFINIF? (Nótese que esta variable se ha escrito en cursiva en el ejemplo anterior debido a que aún no se ha declarado). Es evidente que cada **IF** distinto que aparezca en el programa fuente

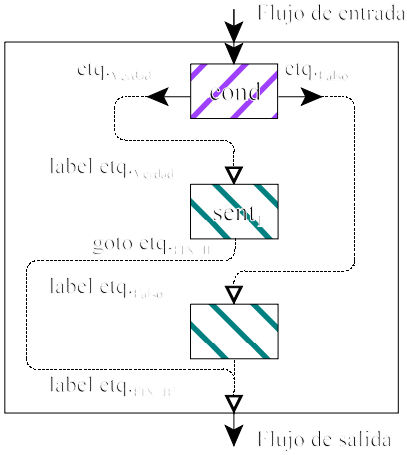


Figure 17 Diagrama de bloques de una sentencia **IF**

necesitará una etiqueta de fin de **IF** diferente. Por tanto, está claro que esta etiqueta no debe almacenarse en una variable global ya que la sentencia que hay en el **ELSE** podría ser otro **IF**, lo que haría necesario reducirlo por esta misma regla y machacaría el valor de la variable global antes de llegar a la acción semántica del final (en negro).

Por otro lado, etqFINIF tampoco puede declararse en la acción semántica en azul (donde se le da valor pasándola como parámetro a la función nuevaEtq), ya que en tal caso se trataría de una variable local a un bloque de lenguaje C y no sería visible en la acción semántica del final.

Por tanto, necesitar declararla en algún lugar tal que sea lo bastante global como para ser vista en dos acciones semánticas de la misma regla; pero también debe ser lo bastante local como para que los **IF** anidados puedan asignar valores locales sin machacar las etiquetas de los **IF** más externos. Es de notar que un metacompilador como JavaCC soluciona este problema mediante el área de código asociado a cada regla BNF; aquí se podría declarar la variable etqFINIF y utilizarla en cualquier acción semántica de esa misma regla. En caso de haber **IF** anidados, la propia recursión en las llamadas a las funciones que representan los no terminales se encargaría de crear nuevos ámbitos con copias de dicha variable.

La solución adoptada por JavaCC nos puede dar alguna pista para solucionar el problema en PCYacc: necesitamos una variable local a la regla completa, que se pueda utilizar en todas sus acciones semánticas, pero que sea diferente si hay **IF** anidados. La solución consiste en utilizar el *token* **IF** como contenedor de dicha variable, asignándosela como atributo: el atributo del *token* **IF** es accesible por las acciones semánticas que hay tras él, y si hay, por ejemplo, tres **IF** anidados, cada uno de ellos tendrá su propio atributo. Asumiendo esto, la regla queda:

```
sent : IF cond THEN        { printf("label %s\n", $2.etqVerdad); }
       sent                { nuevaEtq($1);
```

```
                    printf ("\tgoto %s\n", $1);
                    printf ("label %s:\n", $2.etqFalso); }
opcional  FIN IF    { printf("label %s:\n", $1); }
```

La figura 8.18 muestra el código generado para una sentencia de ejemplo. Nótese que si la parte **ELSE** no existe (el no terminal **opcional** se reduce por la regla épsilon), el código funciona correctamente, a pesar de no ser del todo óptimo.

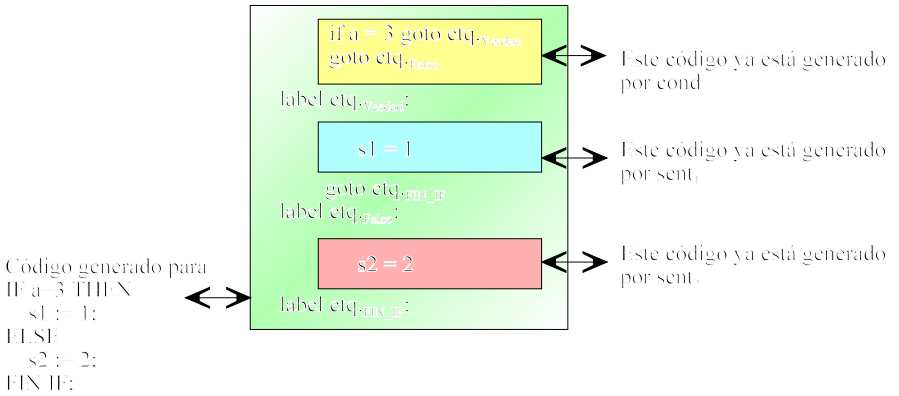


Figure 18Código generado para una sentencia **IF** simple

8.4.4.2 Sentencia WHILE

El caso de **WHILE** y **REPEAT** es muy similar. En ambos es necesario colocar una etiqueta al comienzo del bucle, a la que se saltará para repetir cada iteración.

En el caso de **WHILE**, a continuación se genera el código de la condición, ya que ésta debe evaluarse antes de entrar a ejecutar el cuerpo del bucle. La etiqueta de verdad se ubica justo antes de las sentencias que componen el cuerpo, que es lo que se debe ejecutar si la condición es cierta. Al final de las sentencias se pondrá un salto al inicio del bucle, donde de nuevo se comprobará la condición. La etiqueta de falso de la condición, se ubicará al final de todo lo relacionado con el **WHILE**, o lo que es lo mismo, al principio del código generado para las sentencias que siguen al **WHILE**. De esta manera, cuando la condición deje de cumplirse continuará la ejecución de la secuencia de instrucciones que siguen a la sentencia **WHILE**.

Recordemos que la regla de producción asociada al **WHILE** es:

sent : WHILE cond DO sent FIN WHILE

por lo que los bloques de código de que disponemos para construir adecuadamente nuestro flujo son los que aparecen sombreados en la figura 8.19. El resto de la figura 8.19 muestra dónde deben ubicarse las etiquetas de la condición, así como la necesidad de crear una nueva etiqueta asociada al inicio del **WHILE**. Esta etiqueta tiene

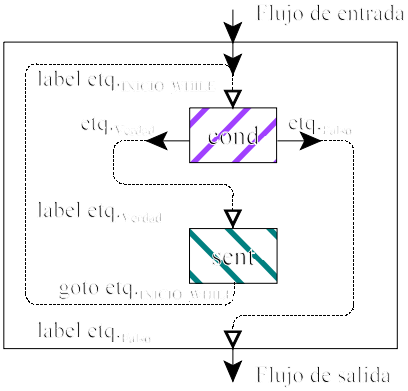


Figure 19 Diagrama de bloques de una sentencia **WHILE**

propiedades muy similares a las ya explicadas en el punto anterior con respecto a la sentencia **IF**. En otras palabras, se almacenará como atributo del propio *token* **WHILE**.

De esta forma, el diagrama de bloques sigue exactamente la semántica de un bucle **WHILE**:

- Se comprueba la condición.
- Si es verdad se ejecuta $sent_1$ y se vuelve al punto anterior, a comprobar la condición.
- Si es falsa se sale del bucle y continúa el flujo por la sentencia que sigue al WHILE.

Finalmente, las acciones semánticas asociadas a la regla de producción del **WHILE** quedarían:

```
sent : WHILE                {
                              nuevaEtq($1);
                              printf("label %s:\n", $1);
                            }
       cond DO              { printf ("label %s:\n", $3.etqVerdad); }
       sent FIN WHILE       {
                              printf("\tgoto %s\n", $1);
                              printf("label %s:\n",$3.etqFalso);
                            }
```

8.4.4.3 Sentencia REPEAT

Recordemos la regla de producción asociada a la sentencia **REPEAT**:

sent : REPEAT sent UNTIL cond

en la que puede observarse cómo la condición se evalúa tras haber ejecutado, al menos, un ciclo. El cuerpo se ejecutará mientras la condición sea falsa, o lo que es lo mismo, hasta que la condición se cumpla.

Para que el código intermedio generado se comporte de la misma manera, lo

ideal es que la etiqueta de falso de la condición se coloque al comienzo del cuerpo, y que la etiqueta de cierto permita continuar el flujo secuencial del programa.

Así, a continuación de la etiqueta de falso se generaría el código de las sentencias, ya que la condición se evalúa al final. Tras las sentencias colocamos el código de la condición. Como la etiqueta de falso ya se ha puesto al comienzo del bucle, sólo queda poner la etiqueta de verdad tras el código de la condición, lo que dará un diagrama de bloques ideal como el que se presenta en la figura 8.20.

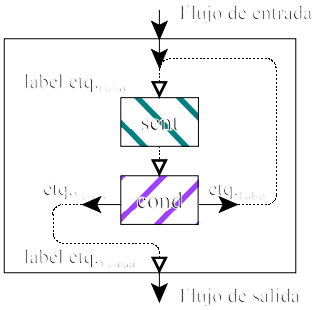


Figure 20 Diagrama de bloques ideal de una sentencia REPEAT

Sin embargo, conseguir generar código que obedezca al diagrama de la figura 8.20 Resulta inviable, toda vez que la etiqueta de falso no se puede colocar antes de las sentencias del cuerpo ya que aún no se conoce su valor. En otras palabras, la etiqueta de falso se genera cuando se reduce la condición, cosa que sucede después de que necesitemos visualizar su **label**. Desde otro punto de vista, cuando se llega a conocer el valor de la etiqueta de falso, es tarde para visualizar su **label**, ya que ello tuvo que hacerse incluso antes de reducir las sentencias del cuerpo. Nótese cómo la siguiente acción es incorrecta porque se intenta acceder a un atributo que se desconoce en el momento de ejecutar dicha acción:

```
sent : REPEAT                { printf ("label %s:\n", $5.etqFalso); }
       sent UNTIL cond
```

Para solucionar este problema hacemos lo que se llama una indirección, o sea, creamos una nueva etiqueta de inicio del **REPEAT** y luego creamos un bloque de código en el que colocamos en secuencia:

- El label de la etiqueta de falso.
- Un goto a la etiqueta de inicio del REPEAT.

El diagrama de bloques quedaría como se ilustra en la figura 8.21. Este código tan ineficiente se podría optimizar en una fase posterior.

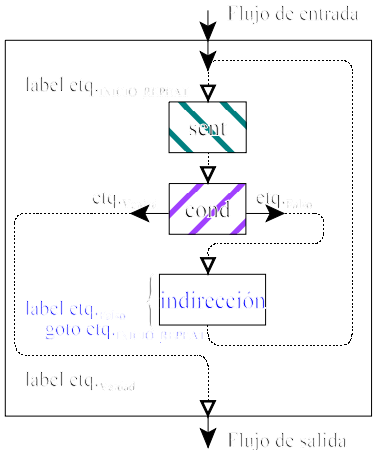


Figure 21 Diagrama de bloques de una sentencia REPEAT

Las acciones semánticas asociadas a la regla de producción del **REPEAT** quedarían:

```
sent : REPEAT                {
                               nuevaEtq($1);
                               printf ("label %s:\n", $1);
                             }
       sent UNTIL cond       {
                               printf ("label %s:\n", $5.etqFalso");
                               printf("\tgoto %s\n", $1);
                               printf ("label %s:\n", $5.etqVerdad);

                             }
```

La figura 8.22 muestra el código generado para una sentencia de ejemplo.

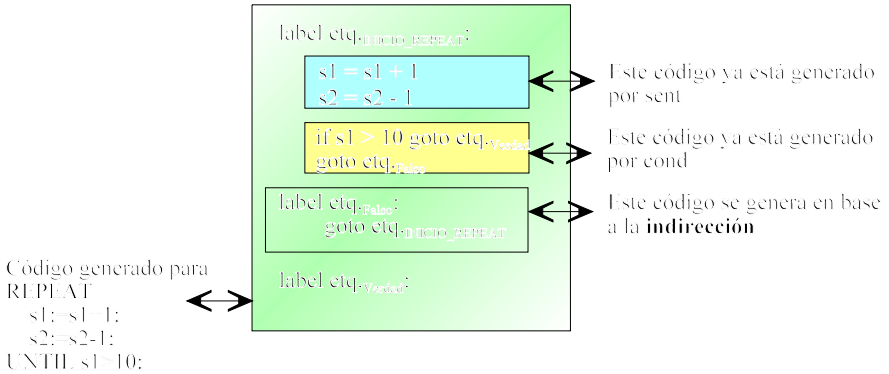


Figure 22Código generado para una sentencia **REPEAT** simple

8.4.4.4 Sentencia CASE

La sentencia **CASE** es la más compleja de traducir, ya que ésta permite un número indeterminado y potencialmente infinito de cláusulas **CASO** en su interior, lo cual obliga a arrastrar una serie de atributos a medida que se van efectuando reducciones. El objetivo semántico de esta sentencia es el mismo que el de la sentencia **switch** del lenguaje C, sólo que no es necesaria una cláusula **break** para romper el flujo al final de la sentencia asociada a cada caso particular.

Por otro lado, y desde el punto de vista gramatical, no podemos utilizar una única regla de producción para indicar su sintaxis por lo que hay que recurrir a reglas recursivas a izquierda y declarar unos cuantos no terminales nuevos. La parte de la gramática que la reconoce es:

```
sent          :  sent_case ;
sent_case     :  inicio_case FIN CASE
              |  inicio_case OTHERWISE sent FIN CASE
              ;
inicio_case   :  CASE expr OF
              |  inicio_case CASO expr ':' sent
              ;
```

Como puede observarse se han incluido dos posibilidades para el no terminal **sent_case**, una con la cláusula **OTHERWISE** y otra sin ella, ya que dicha cláusula es opcional. Ambas reglas podrían haberse sustituído por una sola en la que apareciese un no terminal nuevo que hiciera referencia a la opcionalidad del **OTHERWISE**, como por ejemplo:

```
sent_case       :  inicio_case opcional_case FIN CASE ;
opcional_case   :  OTHERWISE sent
                |  /* Épsilon */
                ;
```

Pero la dejaremos tal y como se ha propuesto para mostrar, junto a la decisión tomada para el IF, cómo una misma cosa se puede hacer de diferentes formas y cómo afecta cada decisión a la hora de incluir las acciones semánticas para generar código intermedio.

Además, la gramática propuesta permite construcciones como "CASE a OF FIN CASE", en la que no se ha incluído ni una sóla cláusula **CASO**. Desde nuestro punto de vista lo consideraremos válido, pero si quisiéramos evitarlo tendríamos que sustituir la primera de **inicio_case** y ponerla como:

```
inicio_case : CASE expr OF CASO expr ':' sent
```

La figura 8.23 muestra parte del árbol sintáctico que reconoce una sentencia **CASE**, en la que se puede apreciar la recursión sobre el no terminal **inicio_case**.

Una vez vistos estos detalles, vamos a pasar ahora a la generación de código intermedio en sí. De lo primero que hay que percatarse es de que la regla principal que

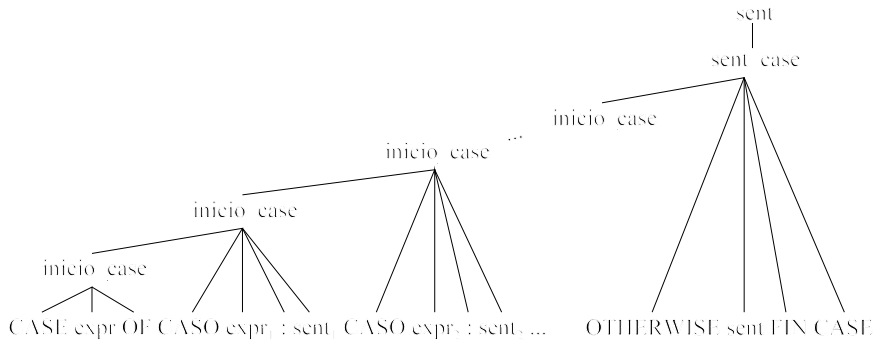


Figure 23Árbol sintáctico que reconoce una sentencia **CASE**

se va a encargar de generar código es la que más aparece en el árbol sintáctico, esto es, la regla recursiva de **inicio_case**:

inicio_case : inicio_case CASO expr ':' sent

de tal manera que la/s acciones semánticas que asociemos a esta regla se ejecutarán repetidas veces durante la reducción de la sentencia **CASE** completa. Teniendo esto en cuenta vamos a dar una idea preliminar de la estructura del código intermedio que queremos generar, considerando que la estructura que controla cada bloque **CASO** debe ser siempre la misma (la estructura, pero no las variables y etiquetas que intervienen en cada **CASO**). Así pues, asociado al ejemplo de la figura 8.23 se tendrían unos bloques como los de la figura 8.24. En esta última figura se han marcado en diferentes color los bloques de código en función de qué regla de producción lo ha generado. En azul están los bloques generados por la segunda regla de **inicio_case**, en verde el generado por la primera regla de **inicio_case**, y en marrón el generado por la regla de **sent_case** que incluye la cláusula **OTHERWISE**. Asimismo, al margen de cada bloque **expr** figura el nombre de la variable (temporal o no) en la que se asume que se almacena el resultado total de evaluar dicha expresión en tiempo de ejecución. Por último, los bloques rayados representan código

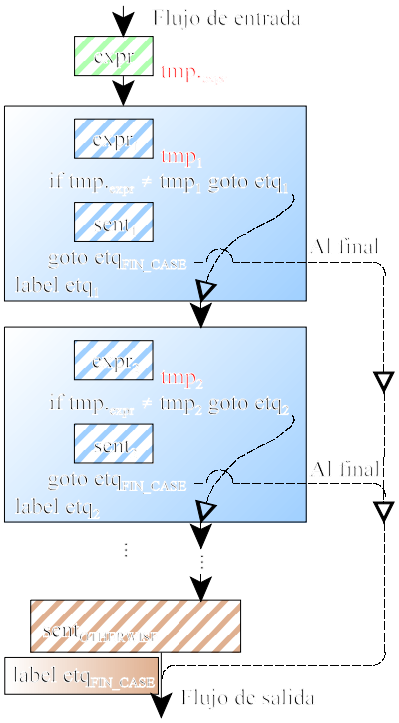


Figure 24 Bloques de código para una sentencia **CASE**

intermedio que ya ha sido generado por otras reglas de producción; por tanto, todo lo que no está rayado debemos generarlo en acciones semánticas asociadas a las reglas de producción del **CASE**.

Como puede verse en esta figura, se ha conseguido que todos los bloques en azul tengan aproximadamente la misma forma:

- el cálculo de una expresión local que se almacena en $\mathbf{tmp_i}$
- una comparación de $\mathbf{tmp._{expr}}$ con $\mathbf{tmp_i}$. Si no son iguales saltamos al final del bloque
- el código intermedio asociado a una sentencia $\mathbf{sent_i}$
- un salto al final de la sentencia **CASE**

Parece claro que la mayor complejidad en la generación de este código, radica en la regla del **CASO** (segunda regla de **inicio_case**), ya que ella es la que debe generar todo el código marcado de azul en la figura 8.24. Estudiemos con detenimiento uno de los bloques azules. Los tercetos que debemos generar en cada bloque de código azul se dividen, a su vez en dos partes: una intercalada entre $\mathbf{expr_i}$ y $\mathbf{sent_i}$, y otra al final del bloque. El siguiente paso a seguir es examinar este código para deducir qué atributos vamos a necesitar en cada no terminal. Centrándonos en las variables y etiquetas que aparecen, podemos clasificarlas en dos apartados:

- La variable $\mathbf{tmp_i}$ y la etiqueta $\mathbf{etq_i}$ son locales a cada bloque, esto es, no se utilizan más que en un solo bloque.
- La variable $\mathbf{tmp._{expr}}$ y la etiqueta $\mathbf{etq_{FIN\_CASE}}$ se utilizan repetidamente en todos los bloques. Además se utiliza en la regla de **sent_case** con el objetivo de visualizar el "**label** $\mathbf{etq_{FIN\_CASE}}$" último. $\mathbf{tmp._{expr}}$ representa a la expresión que se va a ir comparando y $\mathbf{etq_{FIN\_CASE}}$ representa el final del **CASE**, etiqueta a la que se saltará tras poner el código asociado a la sentencia de cada **CASO**.

De un lado, la variable $\mathbf{tmp_i}$ es generada por la expresión local al **CASO** y almacenada en el atributo del no terminal **expr**, por lo que es accesible en cualquier acción semántica de la regla de producción que se ponga detrás de dicho no terminal. En cuanto a la variable $\mathbf{etq_i}$ puede observarse que se utiliza en un **goto** en un terceto entre la generación del código de $\mathbf{expr_i}$ y de $\mathbf{sent_i}$, mientras que el **label** aparece después de $\mathbf{sent_i}$. Esto quiere decir que necesitaremos esta etiqueta en dos acciones semánticas de la misma regla, por lo que optaremos por asociarla como atributo del *token* **CASO**, de forma parecida a como se ha hecho para las sentencias **IF**, **WHILE** y **REPEAT**.

De otro lado, y un poco más complicado, la variable $\mathbf{tmp._{expr}}$ se genera por la **expr** que aparece en la regla del **CASE**, pero debe utilizarse en todas y cada una de las reglas de **CASO** con el objetivo de comparar su valor con las sucesivas $\mathbf{tmp_i}$. Por ello hemos de encontrar algún mecanismo que permita propagar el nombre de esta variable a través de todas las reducciones de la regla del **CASO** necesarias para reconocer la sentencia completa. Con la etiqueta $\mathbf{etq_{FIN\_CASE}}$ sucede algo parecido, ya que dicha etiqueta es necesaria en cada bloque **CASO** con el objetivo de saltar al final del **CASE**

una vez finalizada la ejecución de la sentencia asociada a cada **CASO**. Asimismo, $\mathbf{etq_{FIN\_CASE}}$ debe ser accesible por la regla **sent_case** para colocar el **label** final (en marrón según la figura 8.24).

Para solucionar la accesibilidad de $\mathbf{tmp._{expr}}$ y $\mathbf{etq_{FIN\_CASE}}$ por las acciones semánticas que las necesitan, lo mejor es asociarlas como atributos del no terminal **inicio_case**. Esta decisión es crucial, y supone el mecanismo central por el que daremos solución a la generación de código para la sentencia **CASE**. De esta forma, y tal y como puede apreciarse en la figura 8.25, ambas variables deben propagarse de un no terminal **inicio_case** a otro, haciendo uso de la regla recursiva:

```
inicio_case    :    inicio_case CASO expr ':' sent
```

Así, las acciones semánticas de esta regla deben generar el código intermedio en base a los atributos del **inicio_case** del consecuente y, por último, propagar estos atributos al **inicio_case** del antecedente con el objetivo de que puedan ser utilizados en el siguiente **CASO** (si lo hay). Como puede observarse, el encargado de generar la $\mathbf{etq_{FIN\_CASE}}$ es la primera regla de **inicio_case**, a pesar de que esta regla no va a necesitar dicho atributo: el objetivo es que todas las demás aplicaciones de la regla recursiva dispongan de la misma etiqueta a la que saltar.

Así pues, las reglas de producción y sus acciones semánticas quedan, comenzando por las que primero se reducen:

```
inicio_ case    :  CASE expr OF    {
                                     strcpy ($$.var,$2);
                                     nuevaEtq ($$.etq );
                                   }
          |  inicio_case CASO expr ':'  {
                                          nuevaEtq( $2.etq);
                                          printf ("if %s != %s goto %s",
                                                    $1.var, $3, $2.etq);
                                        }
```

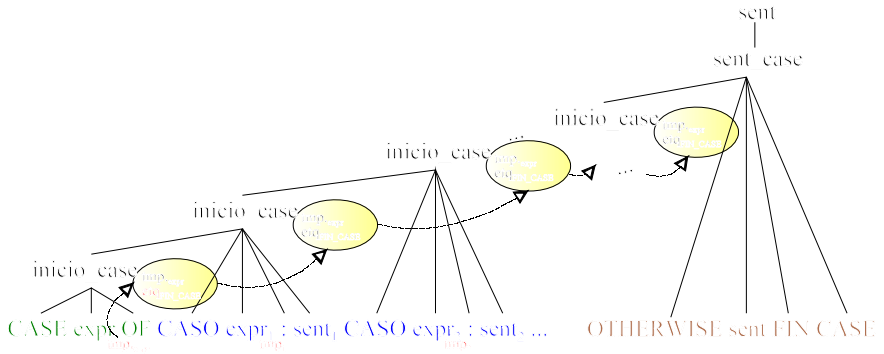


Figure 25 Propagación de atributos del no terminal **inicio_case**. Las cláusulas se han coloreado de acuerdo a la figura 8.24. En rojo se ha indicado la primera aparición de cada atributo

```
                    sent            {
                                      strcpy ($$.var, $1.var);
                                      strcpy( $$.etq, $1.etq);
                                      printf("goto %s", $2.etq);
                                      printf("label %s", $2.etq);
                                    }
                ;
sent_ case      : inicio_case OTHERWISE sent FIN CASE { printf("label %s", $1.etq); }
                | inicio_case FIN CASE { printf("label %s", $1.etq); }
                ;
```

8.4.5 Solución con Lex/Yacc

A continuación se expone todo el código explicado en los apartados anteriores. Comenzamos con el código Lex cuyo único objetivo es reconocer las palabras reservadas, los identificadores y los números, así como ignorar comentarios y llevar la cuenta del número de línea por el que avanza el análisis léxico. Los comentarios se extienden en líneas completas y comienzan por un asterisco:

```
1  %{
2        int linea_actual = 1;
3  %}
4  %START COMENT
5  %%
6  ^[ \t]*"*"        { BEGIN COMENT; }
7  <COMENT>.+        { ; }
8  <COMENT>\n        { BEGIN 0; linea_actual ++; }
9
10 ":="        { return ASIG; }
11 ">="        { return MAI; }
12 "<="        { return MEI; }
13 "!="        { return DIF; }
14 CASE        { return CASE; }
15 OF          { return OF; }
16 CASO        { return CASO; }
17 OTHERWISE       { return OTHERWISE; }
18 REPEAT      { return REPEAT; }
19 UNTIL       { return UNTIL; }
20 IF          { return IF; }
21 THEN        { return THEN; }
22 ELSE        { return ELSE; }
23 WHILE       { return WHILE; }
24 DO          { return DO; }
25 AND         { return AND; }
26 OR          { return OR; }
27 NOT         { return NOT; }
28 FIN         { return FIN; }
29
30 [0-9]+      {
31                   strcpy(yylval.numero, yytext);
```

```
32                      return NUMERO;
33              }
34  [A-Za-z_][A-Za-z0-9_]*      {
35                                      strcpy(yylval.variable_aux, yytext);
36                                      return ID;
37                                      }
38  [ \t]+      { ; }
39  \n          { linea_actual++; }
40  .           { return yytext[0]; }
```

En cuanto al código Yacc, es el siguiente:

```
 1  /* Declaraciones de apoyo */
 2  %{
 3  typedef struct _doble_cond {
 4          char        etq_verdad[21],
 5                      etq_falso[21];
 6  } doble_cond;
 7  typedef struct _datos_case {
 8          char        etq_final[21];
 9          char        variable_expr[21];
10  } datos_case;
11  %}
12  /* Declaracion de atributos */
13  %union {
14          char numero[21];
15          char variable_aux[21];
16          char etiqueta_aux[21];
17          char etiqueta_siguiente[21];
18          doble_cond bloque_cond;
19          datos_case bloque_case;
20  }
21  /* Declaracion de tokens y sus atributos */
22  %token <numero> NUMERO
23  %token <variable_aux> ID
24  %token <etiqueta_aux> IF WHILE REPEAT
25  %token <etiqueta_siguiente> CASO
26  %token ASIG THEN ELSE FIN DO UNTIL CASE OF OTHERWISE
27  %token MAI MEI DIF
28  /* Declaración de no terminales y sus atributos */
29  %type <variable_aux> expr
30  %type <bloque_cond> cond
31  %type <bloque_case> inicio_case
32  /* Precedencia y asociatividad de operadores */
33  %left OR
34  %left AND
35  %left NOT
36  %left '+' '-'
37  %left '*' '/'
38  %left MENOS_UNARIO
39
```

```
40 %%
41 prog      :     prog sent ';'
42           |     prog error ';' { yyerrok; }
43           |
44           ;
45 sent      :     ID ASIG expr      {
46                                       printf("\t%s = %s\n", $1, $3);
47                                     }
48           |     IF cond {
49                           printf("label %s\n", $2.etq_verdad);
50                         }
51                 THEN sent ';' {
52                                   nuevaEtq($1);
53                                   printf("\tgoto %s\n", $1);
54                                   printf("label %s\n", $2.etq_falso);
55                                 }
56                 opcional
57                 FIN IF    {
58                           printf("label %s\n", $1);
59                         }
60           |     '{' lista_sent '}' { ; }
61           |     WHILE  {
62                           nuevaEtq($1);
63                           printf("label %s\n", $1);
64                         }
65                 cond     {
66                           printf("label %s\n", $3.etq_verdad);
67                         }
68                 DO sent ';'
69                 FIN WHILE   {
70                                   printf("\tgoto %s\n", $1);
71                                   printf("label %s\n", $3.etq_falso);
72                                 }
73           |     REPEAT      {
74                                   nuevaEtq($1);
75                                   printf("label %s\n", $1);
76                                 }
77                 sent ';'
78                 UNTIL cond  {
79                                   printf("label %s\n", $6.etq_falso);
80                                   printf("\tgoto %s\n", $1);
81                                   printf("label %s\n", $6.etq_verdad);
82                                 }
83           |     sent_case
84           ;
85 opcional :     ELSE sent ';'
86                 |     /* Epsilon */
87                 ;
88 lista_sent      :     /* Epsilon */
```

```
 89                 |    lista_sent sent ';'
 90                 |    lista_sent error ';' { yyerrok; }
 91                 ;
 92 sent_case       :    inicio_case
 93                      OTHERWISE sent ';'
 94                      FIN CASE {
 95                                  printf("label %s\n", $1.etq_final);
 96                               }
 97                 |    inicio_case
 98                      FIN CASE {
 99                                  printf("label %s\n", $1.etq_final);
100                               }
101                 ;
102 inicio_case     :    CASE expr OF    {
103                                        strcpy($$.variable_expr, $2);
104                                        nuevaEtq($$.etq_final);
105                                      }
106                 |    inicio_case
107                      CASO expr ':' {
108                                        nuevaEtq($2);
109                                        printf("\tif %s != %s goto %s\n",
110                                                $1.variable_expr, $3, $2);
111                                     }
112                      sent ';'       {
113                                        printf("\tgoto %s\n", $1.etq_final);
114                                        printf("label %s\n", $2);
115                                        strcpy($$.variable_expr, $1.variable_expr);
116                                        strcpy($$.etq_final, $1.etq_final);
117                                     }
118                 ;
119 expr       :    NUMERO    { generarTerceto("\t%s = %s\n", $$, $1, NULL); }
120            |    ID         { strcpy($$, $1); }
121            |    expr '+' expr { generarTerceto("\t%s = %s + %s\n", $$, $1, $3); }
122            |    expr '-' expr  { generarTerceto("\t%s = %s - %s\n", $$, $1, $3); }
123            |    expr '*' expr  { generarTerceto("\t%s = %s * %s\n", $$, $1, $3); }
124            |    expr '/' expr  { generarTerceto("\t%s = %s / %s\n", $$, $1, $3); }
125            |    '-' expr %prec MENOS_UNARIO
126                                { generarTerceto("\t%s = - %s\n", $$, $2, NULL); }
127            |    '(' expr ')'   { strcpy($$, $2); }
128            ;
129 cond       :    expr '>' expr       { generarCondicion($1, ">", $3, &($$)); }
130            |    expr '<' expr       { generarCondicion($1, "<", $3, &($$)); }
131            |    expr MAI expr       { generarCondicion($1, ">=", $3, &($$)); }
132            |    expr MEI expr       { generarCondicion($1, "<=", $3, &($$)); }
133            |    expr '=' expr       { generarCondicion($1, "=", $3, &($$)); }
134            |    expr DIF expr       { generarCondicion($1, "!=", $3, &($$)); }
135            |    NOT cond            { strcpy($$.etq_verdad, $2.etq_falso);
136                                       strcpy($$.etq_falso, $2.etq_verdad);
137                                     }
```

```
138          |      cond AND    {
139                                 printf("label %s\n", $1.etq_verdad);
140                              }
141                 cond         {
142                                 printf("label %s\n", $1.etq_falso);
143                                 printf("\tgoto %s\n", $4.etq_falso);
144                                 strcpy($$.etq_verdad, $4.etq_verdad);
145                                 strcpy($$.etq_falso, $4.etq_falso);
146                              }
147          |      cond OR    {
148                                 printf("label %s\n", $1.etq_falso);
149                              }
150                 cond         {
151                                 printf("label %s\n", $1.etq_verdad);
152                                 printf("\tgoto %s\n", $4.etq_verdad);
153                                 strcpy($$.etq_verdad, $4.etq_verdad);
154                                 strcpy($$.etq_falso, $4.etq_falso);
155                              }
156          |      '(' cond ')'  {
157                                 strcpy($$.etq_verdad, $2.etq_verdad);
158                                 strcpy($$.etq_falso, $2.etq_falso);
159                              }
160          ;
161
162  %%
163  #include "ejem6l.c"
164  void main() {
165     yyparse();
166  }
167  void yyerror(char * s) {
168     fprintf(stderr, "Error de sintaxis en la linea %d\n", linea_actual);
169  }
170  void nuevaTmp(char * s) {
171    static actual=0;
172     sprintf(s, "tmp%d", ++actual);
173  }
174  void nuevaEtq(char * s) {
175    static actual=0;
176     sprintf(s, "etq%d", ++actual);
177  }
178  void generarTerceto(       char * terceto,
179                     char * Lvalor,
180                     char * Rvalor1,
181                     char * Rvalor2){
182        nuevaTmp(Lvalor);
183        printf(terceto, Lvalor, Rvalor1, Rvalor2);
184  }
185  void generarCondicion(   char * Rvalor1,
186                     char * condicion,
```

```
187                 char * Rvalor2,
188                 doble_cond * etqs){
189         nuevaEtq((*etqs).etq_verdad);
190         nuevaEtq((*etqs).etq_falso);
191         printf("\tif %s %s %s goto %s\n", Rvalor1, condicion, Rvalor2, (*etqs).etq_verdad);
192         printf("\tgoto %s\n", (*etqs).etq_falso);
193 }
```

Para empezar, la generación de los tercetos más frecuentes (expresiones aritméticas y condiciones simples) se ha delegado en las funciones generarTerceto y GenerarCondicion de las líneas 178 a 193, que reciben los parámetros adecuados y hacen que las reglas de las líneas 119 a 134 queden mucho más compactas y claras.

El %union de la línea 13 junto a las declaraciones de tipos de las líneas precedentes establece el marco de atributos para los terminales y no terminales de la gramática. Es de notar que el %union posee varios campos con el mismo tipo (char[21]), pero distinto nombre: numero, variable_aux, etiqueta_aux y etiqueta_siguiente, al igual que dos registros con la misma estructura interna (dos cadenas de 21 caracteres): bloque_cond y bloque_case. Ello se debe a que, aunque estructuralmente coincidan, semánticamente tienen objetivos distintos, por lo que se ha optado por replicarlos con nombres distintivos. Esto no tiene ninguna repercusión desde el punto de vista de la eficiencia, puesto que el espacio ocupado por el %union es el mismo.

Por último, como atributo del *token* **NUMERO** se ha escogido un campo de tipo texto en lugar de un valor numérico entero, ya que nuestro propósito es generar tercetos y no operar aritméticamente con los valores en sí. Además, la acción semántica de la línea 119 obliga a que todo literal numérico sea gestionado mediante una variable temporal merced a un terceto de asignación directa.

8.4.6 Solución con JFlex/Cup

La solución con JFlex y Cup es muy similar a la dada anteriormente para Lex y Yacc. La única diferencia relevante con respecto a JFlex radica en que debe ser éste quien se encargue de asignar las etiquetas necesarias a los terminales **IF**, **WHILE**, **REPEAT** y **CASO** debido a que los atributos no se pueden modificar en las acciones intermedias de Cup. Para dejar esto más patente el código JFlex tiene una función estática que permite generar etiquetas (con el prefijo **etqL** para distinguirlas de las generadas por Cup que llevarán el prefijo **etqY**), ya que las acciones de JFlex no pueden acceder a las definiciones de funciones declaradas en Cup. Así pues el código queda:

```
1 import java_cup.runtime.*;
2 import java.io.*;
3 %%
4 %{
5       int lineaActual = 1;
6       private static int actualEtq=0;
7       private static String nuevaEtq() {
```

```
 8              return "etqL"+(++actualEtq);
 9          }
10  %}
11  %unicode
12  %cup
13  %line
14  %column
15  %state COMENT
16  %%
17  ^[ \t]*"*"                  { yybegin(COMENT); }
18  <COMENT>.+      { ; }
19  <COMENT>\n      { lineaActual ++; yybegin(YYINITIAL); }
20  "+"   { return new Symbol(sym.MAS); }
21  "*"   { return new Symbol(sym.POR); }
22  "/"   { return new Symbol(sym.ENTRE); }
23  "-"   { return new Symbol(sym.MENOS); }
24  "("   { return new Symbol(sym.LPAREN); }
25  ")"   { return new Symbol(sym.RPAREN); }
26  "{"   { return new Symbol(sym.LLLAVE); }
27  "}"   { return new Symbol(sym.RLLAVE); }
28  ";"   { return new Symbol(sym.PUNTOYCOMA); }
29  ":"   { return new Symbol(sym.DOSPUNTOS); }
30  ":="  { return new Symbol(sym.ASIG); }
31  ">"   { return new Symbol(sym.MAYOR); }
32  "<"   { return new Symbol(sym.MENOR); }
33  "="   { return new Symbol(sym.IGUAL); }
34  ">="        { return new Symbol(sym.MAI); }
35  "<="        { return new Symbol(sym.MEI); }
36  "!="        { return new Symbol(sym.DIF); }
37  CASE    { return new Symbol(sym.CASE); }
38  OF      { return new Symbol(sym.OF); }
39  CASO    { return new Symbol(sym.CASO, nuevaEtq()); }
40  OTHERWISE       { return new Symbol(sym.OTHERWISE); }
41  REPEAT          { return new Symbol(sym.REPEAT, nuevaEtq()); }
42  UNTIL   { return new Symbol(sym.UNTIL); }
43  IF      { return new Symbol(sym.IF, nuevaEtq()); }
44  THEN    { return new Symbol(sym.THEN); }
45  ELSE    { return new Symbol(sym.ELSE); }
46  WHILE   { return new Symbol(sym.WHILE, nuevaEtq()); }
47  DO      { return new Symbol(sym.DO); }
48  AND     { return new Symbol(sym.AND); }
49  OR      { return new Symbol(sym.OR); }
50  NOT     { return new Symbol(sym.NOT); }
51  FIN     { return new Symbol(sym.FIN); }
52  [:jletter:][:jletterdigit:]*  { return new Symbol(sym.ID, yytext()); }
53  [:digit:]+                  { return new Symbol(sym.NUMERO, yytext()); }
54  [ \t\r]+    {;}
55  [\n]        { lineaActual++; }
56  .           { System.out.println("Error léxico en línea "+lineaActual+":-"+yytext()+"-"); }
```

Con respecto a Cup, la diferencia fundamental radica en la utilización de objetos pertenecientes a las clases DatosCASE y BloqueCondicion para poder asignar más de un atributo a algunos no terminales. Por otro lado, también es de notar que la funciones generaTerceto y generaCondicion se han adaptado a la filosofía Java y, a diferencia de Lex y Yacc, ahora generan un atributo que se debe asignar a RESULT en la invocación. Por último, los nombres de atributos *in situ* (indicados en la propia regla y separados del símbolo a que pertenecen mediante dos puntos) han sido escogidos con cuidado para hacer el código más legible. El programa Cup queda:

```
1  import java_cup.runtime.*;
2  import java.io.*;
3  parser code {:
4      public static void main(String[] arg){
5          Yylex miAnalizadorLexico = new Yylex(new InputStreamReader(System.in));
6          parser parserObj = new parser(miAnalizadorLexico);
7          try{
8              parserObj.parse();
9          }catch(Exception x){
10             x.printStackTrace();
11             System.out.println("Error fatal.");
12         }
13     }
14 :};
15 action code {:
16     class DatosCASE {
17         String tmpExpr, etqFinal;
18     }
19     class BloqueCondicion {
20         String etqVerdad, etqFalso;
21     }
22     private static int actualTmp=0;
23     private static String nuevaTmp() {
24         return "tmp"+(++actualTmp);
25     }
26     private static int actualEtq=0;
27     private static String nuevaEtq() {
28         return "etqY"+(++actualEtq);
29     }
30     private String generarTerceto(String terceto) {
31         String tmp = nuevaTmp();
32         System.out.println(tmp + terceto);
33         return tmp;
34     }
35     private BloqueCondicion generarCondicion(  String Rvalor1,
36                                                String condicion,
37                                                String Rvalor2) {
38         BloqueCondicion etqs = new BloqueCondicion();
39         etqs.etqVerdad = nuevaEtq();
40         etqs.etqFalso = nuevaEtq();
```

```
41              System.out.println("\tif "+ Rvalor1 + condicion + Rvalor2
42                                                      +" goto "+ etqs.etqVerdad);
43              System.out.println("\tgoto "+ etqs.etqFalso);
44              return etqs;
45          }
46  :}
47  terminal PUNTOYCOMA, DOSPUNTOS, MAS, POR, ENTRE, MENOS, UMENOS;
48  terminal LPAREN, RPAREN, LLLAVE, RLLAVE;
49  terminal MAYOR, MENOR, IGUAL, MAI, MEI, DIF;
50  terminal AND, OR, NOT;
51  terminal String ID, NUMERO;
52  terminal String IF, WHILE, REPEAT, CASO;
53  terminal ASIG, THEN, ELSE, FIN, DO, UNTIL, CASE, OF, OTHERWISE;
54  non terminal String expr;
55  non terminal BloqueCondicion cond;
56  non terminal DatosCASE inicioCASE;
57  non terminal prog, sent, sentCASE, opcional, listaSent;
58  precedence left OR;
59  precedence left AND;
60  precedence right NOT;
61  precedence left MAS, MENOS;
62  precedence left POR, ENTRE;
63  precedence right UMENOS;
64  /* Gramática */
65  prog        ::=   prog sent PUNTOYCOMA
66                |   prog error PUNTOYCOMA
67                |
68                ;
69  sent        ::=   ID:id ASIG expr:e       {:
70                        System.out.println("\t"+ id +" = "+ e);
71                      :}
72                |   IF:etqFinIf cond:c       {:
73                        System.out.println("label "+ c.etqVerdad);
74                      :}
75                    THEN sent PUNTOYCOMA {:
76                        System.out.println("\tgoto "+ etqFinIf);
77                        System.out.println("label "+ c.etqFalso);
78                      :}
79                    opcional
80                    FIN IF    {:
81                        System.out.println("label "+ etqFinIf);
82                      :}
83
84                |   LLLAVE listaSent RLLAVE {: ; :}
85                |   WHILE:etqInicioWhile       {:
86                        System.out.println("label "+ etqInicioWhile);
87                      :}
88                    cond:c   {:
89                        System.out.println("label "+ c.etqVerdad);
```

```
90                      :}
91                  DO sent PUNTOYCOMA
92                  FIN WHILE {:
93                        System.out.println("\tgoto "+ etqInicioWhile);
94                        System.out.println("label "+ c.etqFalso);
95                      :}
96              |   REPEAT:etqInicioRepeat  {:
97                        System.out.println("label "+ etqInicioRepeat);
98                      :}
99                  sent PUNTOYCOMA
100                 UNTIL cond:c {:
101                       System.out.println("label "+ c.etqFalso);
102                       System.out.println("\tgoto "+ etqInicioRepeat);
103                       System.out.println("label "+ c.etqVerdad);
104                     :}
105             |   sentCASE
106             ;
107 opcional ::=   ELSE sent PUNTOYCOMA
108             |   /* Epsilon */
109             ;
110 listaSent   ::= /* Epsilon */
111             |   listaSent sent PUNTOYCOMA
112             |   listaSent error PUNTOYCOMA
113             ;
114 sentCASE    ::= inicioCASE:ic OTHERWISE sent PUNTOYCOMA
115                 FIN CASE {:
116                       System.out.println("label "+ ic.etqFinal);
117                     :}
118             |   inicioCASE:ic
119                 FIN CASE {:
120                       System.out.println("label "+ ic.etqFinal);
121                     :}
122             ;
123 inicioCASE  ::= CASE expr:e OF {:
124                       RESULT = new DatosCASE();
125                       RESULT.tmpExpr = e.nombreVariable;
126                       RESULT.etqFinal = nuevaEtq();
127                     :}
128             |   inicioCASE:ic
129                 CASO:etqFinCaso expr:e DOSPUNTOS {:
130                       System.out.println("\tif "+ ic.tmpExpr +" != "+ e
131                                                   +" goto "+ etqFinCaso);
132                     :}
133                 sent PUNTOYCOMA {:
134                       System.out.println("\tgoto "+ ic.etqFinal);
135                       System.out.println("label "+ etqFinCaso);
136                       RESULT = ic;
137                     :}
138             ;
```

```
139 expr ::= ID:id                  {: RESULT = id; :}
140       |  NUMERO:n               {: RESULT = generarTerceto(" = "+ n); :}
141       |  expr:e1 MAS expr:e2  {: RESULT = generarTerceto(" = "+ e1 +" + "+ e2); :}
142       |  expr:e1 MENOS expr:e2 {: RESULT=generarTerceto(" = "+ e1 +" - "+ e2); :}
143       |  expr:e1 POR expr:e2   {: RESULT = generarTerceto(" = "+ e1 +" * "+ e2); :}
144       |  expr:e1 ENTRE expr:e2 {: RESULT =generarTerceto(" = "+ e1 +" / "+ e2); :}
145       |  MENOS expr:e1         {: RESULT = generarTerceto(" = -"+ e1); :}
146                   %prec UMENOS
147       |  LPAREN expr:e1 RPAREN       {: RESULT = e1; :}
148       ;
149 cond  ::=   expr:e1 MAYOR expr:e2  {: RESULT = generarCondicion(e1, ">", e2); :}
150       |      expr:e1 MENOR expr:e2  {: RESULT = generarCondicion(e1, "<", e2); :}
151       |      expr:e1 MAI expr:e2       {: RESULT = generarCondicion(e1, ">=", e2); :}
152       |      expr:e1 MEI expr:e2       {: RESULT = generarCondicion(e1, "<=", e2); :}
153       |      expr:e1 IGUAL expr:e2   {: RESULT = generarCondicion(e1, "=", e2); :}
154       |      expr:e1 DIF expr:e2       {: RESULT = generarCondicion(e1, "!=", e2); :}
155       |      NOT cond:c      {:
156                               RESULT = new BloqueCondicion();
157                               RESULT.etqVerdad = c.etqFalso;
158                               RESULT.etqFalso = c.etqVerdad;
159                              :}
160       |      cond:c1 AND {:
161                               System.out.println("label "+ c1.etqVerdad);
162                              :}
163              cond:c2     {:
164                               System.out.println("label "+ c1.etqFalso);
165                               System.out.println("\tgoto "+ c2.etqFalso);
166                               RESULT = c2;
167                              :}
168       |      cond:c1 OR      {:
169                               System.out.println("label "+ c1.etqFalso);
170                              :}
171              cond:c2     {:
172                               System.out.println("label "+ c1.etqVerdad);
173                               System.out.println("\tgoto "+ c2.etqVerdad);
174                               RESULT = c2;
175                              :}
176       |      LPAREN cond:c1 RPAREN    {: RESULT = c1; :}
177       ;
```

8.4.7 Solución con JavaCC

La solución con JavaCC es muy similar a la dada en JFlex y Cup, sólo que se aprovechan las características de la notación BNF para ahorrar algunos atributos; por ejemplo, ya no es necesaria la clase **DatosCASE** pues en una sola regla se dispone de toda la información necesaria.

Por desgracia, JavaCC no posee una mínima detección de sensibilidad al contexto en su analizador lexicográfico, por lo que detectar los comentarios al inicio

de la línea pasa por utilizar estado léxicos que complican un poco la notación, y que obligan a declarar todos y cada uno de los *tokens* de la gramática con el objetivo de pasar al estado por defecto (**DEFAULT**) una vez reconocido cada uno de ellos. Esto provoca, además, que haya que comenzar el análisis lexicográfico con el estado léxico INICIO_LINEA activado, lo que se encarga de hacer la función main() a través de la función SwitchTo de la clase TokenManager.

Así, la solución completa es:

```
1  PARSER_BEGIN(Control)
2      import java.util.*;
3      public class Control{
4
5          private static class BloqueCondicion{
6              String etqVerdad, etqFalso;
7          }
8          public static void main(String args[]) throws ParseException {
9              ControlTokenManager tm =
10                     new ControlTokenManager(new SimpleCharStream(System.in));
11             tm.SwitchTo(tm.INICIO_LINEA);
12             new Control(tm).gramatica();
13         }
14         private static int actualTmp=0, actualEtq=0;
15         private static String nuevaTmp(){
16             return "tmp"+(++actualTmp);
17         }
18         private static String nuevaEtq(){
19             return "etq"+(++actualEtq);
20         }
21         private static void usarASIG(String s, String e){
22             System.out.println(s+"="+e);
23         }
24         private static String usarOpAritmetico(String e1, String e2, String op){
25             String tmp = nuevaTmp();
26             System.out.println("\t"+tmp+"="+e1+op+e2);
27             return tmp;
28         }
29         private static void usarLabel(String label){
30             System.out.println("label "+ label);
31         }
32         private static void usarGoto(String label){
33             System.out.println("\tgoto "+ label);
34         }
35         private static BloqueCondicion usarOpRelacional(String e1,
36                                                         String e2,
37                                                         String op){
38             BloqueCondicion blq = new BloqueCondicion();
39             blq.etqVerdad = nuevaEtq();
40             blq.etqFalso = nuevaEtq();
```

```
41                  System.out.println("\tif "+ e1+op+e2 +" goto "+ blq.etqVerdad);
42                  usarGoto(blq.etqFalso);
43                  return blq;
44              }
45              private static void intercambiarCondicion(BloqueCondicion blq){
46                  String aux = blq.etqVerdad;
47                  blq.etqVerdad = blq.etqFalso;
48                  blq.etqFalso = blq.etqVerdad;
49              }
50          }
51  PARSER_END(Control)
52  <DEFAULT, INICIO_LINEA> SKIP : {
53              " "
54          |   "\t"
55          |   "\r"
56          |   "\n" : INICIO_LINEA
57  }
58  <INICIO_LINEA> SKIP: {
59          <COMENTARIO: "*" (~["\n"])* "\n">
60  }
61  <DEFAULT, INICIO_LINEA> TOKEN [IGNORE_CASE] : {
62              <NUMERO: (["0"-"9"])+>    : DEFAULT
63          |   <PUNTOYCOMA: ";">         : DEFAULT
64  }
65  <DEFAULT, INICIO_LINEA> TOKEN : {
66              <ASIG: ":=">                    : DEFAULT
67          |   <DOSPUNTOS: ":">                : DEFAULT
68          |   <MAS: "+">                      : DEFAULT
69          |   <POR: "*">                      : DEFAULT
70          |   <ENTRE: "/">                    : DEFAULT
71          |   <MENOS: "-">                    : DEFAULT
72          |   <LPAREN: "(">                   : DEFAULT
73          |   <RPAREN: ")">                   : DEFAULT
74          |   <LLLAVE: "{">                   : DEFAULT
75          |   <RLLAVE: "}">                   : DEFAULT
76          |   <LCOR: "[">                     : DEFAULT
77          |   <RCOR: "]">                     : DEFAULT
78          |   <MAYOR: ">">                    : DEFAULT
79          |   <MENOR: "<">                    : DEFAULT
80          |   <IGUAL: "=">                    : DEFAULT
81          |   <MAI: ">=">                     : DEFAULT
82          |   <MEI: "<=">                     : DEFAULT
83          |   <DIF: "!=">                     : DEFAULT
84          |   <AND: "AND">                    : DEFAULT
85          |   <OR: "OR">                      : DEFAULT
86          |   <NOT: "NOT">                    : DEFAULT
87          |   <IF: "IF">                      : DEFAULT
88          |   <WHILE: "WHILE">                : DEFAULT
89          |   <REPEAT: "REPEAT">              : DEFAULT
```

```
90      |   <CASO: "CASO">                  : DEFAULT
91      |   <THEN: "THEN">                  : DEFAULT
92      |   <ELSE: "ELSE">                  : DEFAULT
93      |   <FIN: "FIN">                    : DEFAULT
94      |   <DO: "DO">                      : DEFAULT
95      |   <UNTIL: "UNTIL">                : DEFAULT
96      |   <CASE: "CASE">                  : DEFAULT
97      |   <OF: "OF">                      : DEFAULT
98      |   <OTHERWISE: "OTHERWISE">        : DEFAULT
99  }
100 <DEFAULT, INICIO_LINEA> TOKEN [IGNORE_CASE] : {
101         <ID: ["A"-"Z", "_"](["A"-"Z", "0"-"9", "_"])*> : DEFAULT
102 }
103 <DEFAULT, INICIO_LINEA> SKIP : {
104     <ILEGAL: (~[])>   { System.out.println("Carácter: "+image+" no esperado.");}
105                       : DEFAULT
106 }
107 /*
108 gramatica ::= ( sentFinalizada )*
109 */
110 void gramatica():{}{
111     (sentFinalizada())*
112 }
113 /*
114 sentFinalizada ::=      ID ASIG expr ';'
115         |   IF cond THEN sentFinalizada [ ELSE sentFinalizada ] FIN IF ';'
116         |   WHILE cond DO sentFinalizada FIN WHILE ';'
117         |   REPEAT sentFinalizada UNTIL cond ';'
118         |   '{' ( sentFinalizada )* '}'
119         |   CASE expr OF ( CASO expr ':' sentFinalizada )*
120                             [ OTHERWISE sentFinalizada ] FIN CASE ';'
121         |   error ';'
122 */
123 void sentFinalizada():{
124     String s;
125     String e, ei;
126     BloqueCondicion c;
127     String etqFinIf, etqInicioWhile, etqInicioRepeat, etqFinalCaso, etqFinCase;
128 }{
129     try {
130     (
131         s=id() <ASIG> e=expr()      { System.out.println("\t"+s+"="+e); }
132     |   <IF> c=cond() <THEN>        { usarLabel(c.etqVerdad); }
133             sentFinalizada()        {
134                                       usarGoto(etqFinIf=nuevaEtq());
135                                       usarLabel(c.etqFalso);
136                                     }
137             [<ELSE> sentFinalizada()]
138         <FIN> <IF>                  { usarLabel(etqFinIf); }
```

```
139     |   <WHILE>                 { usarLabel(etqInicioWhile=nuevaEtq()); }
140             c=cond() <DO>       { usarLabel(c.etqVerdad); }
141             sentFinalizada()
142         <FIN> <WHILE>           {
143                                   usarGoto(etqInicioWhile);
144                                   usarLabel(c.etqFalso);
145                                 }
146     |   <REPEAT>          { usarLabel(etqInicioRepeat=nuevaEtq()); }
147             sentFinalizada()
148         <UNTIL> c=cond() {
149                             usarLabel(c.etqFalso);
150                             usarGoto(etqInicioRepeat);
151                             usarLabel(c.etqVerdad);
152                           }
153     |   <LLLAVE> gramatica() <RLLAVE>
154     |   <CASE> e=expr() <OF>    { etqFinCase = nuevaEtq(); }
155             (<CASO> ei=expr() <DOSPUNTOS>     {
156                                                 System.out.println("\tif "+ e+"!="+ei
157                                                  +"goto "+ (etqFinalCaso=nuevaEtq()));
158                                                         }
159                 sentFinalizada()   {
160                                      usarGoto(etqFinCase);
161                                      usarLabel(etqFinalCaso);
162                                    }
163             )*
164             [<OTHERWISE> sentFinalizada()]
165         <FIN> <CASE>                { usarLabel(etqFinCase); }
166     ) <PUNTOYCOMA>
167     }catch(ParseException x){
168         System.out.println(x.toString());
169         Token t;
170         do {
171             t = getNextToken();
172         } while (t.kind != PUNTOYCOMA);
173     }
174 }
175 /*
176 expr ::= term (('+'|'-') term)*
177 */
178 String expr():{
179     String t1, t2;
180 }{
181     t1=term()     ( <MAS>   t2=term()    { t1=usarOpAritmetico(t1, t2, "+"); }
182                   | <MENOS> t2=term() { t1=usarOpAritmetico(t1, t2, "-"); }
183                   )*    { return t1; }
184 }
185 /*
186 term ::= fact (('*'|'/') fact)*
187 */
```

```
188 String term():{
189       String f1, f2;
190 }{
191       f1=fact()       ( <POR>   f2=fact()     { f1=usarOpAritmetico(f1, f2, "*"); }
192                       | <ENTRE> f2=fact()    { f1=usarOpAritmetico(f1, f2, "/"); }
193                       )*    { return f1; }
194 }
195 /*
196 fact ::= ('-')* ( ID | NUMERO | '(' expr ')' )
197 */
198 String fact():{
199       String s, e, temporal;
200       boolean negado = false;
201 }{    (<MENOS> {negado = !negado;} )*
202       (
203             s=id()          { temporal = s; }
204       |     s=numero()      { temporal = s; }
205       |     <LPAREN> e=expr() <RPAREN> { temporal = e; }
206       ) { if (negado) temporal=usarOpAritmetico("", temporal, "-");
207           return temporal;
208         }
209 }
210 /*
211 cond ::= condTerm (OR condTerm)*
212 */
213 BloqueCondicion cond():{
214       BloqueCondicion c1, c2;
215 }{
216       c1=condTerm()   (     <OR>    { System.out.println("label "+ c1.etqFalso); }
217                             c2=condTerm()   {
218                                   System.out.println("label "+ c1.etqVerdad);
219                                   System.out.println("\tgoto "+ c2.etqVerdad);
220                                   c1 = c2;
221                                         }
222                       )* { return c1; }
223 }
224 /*
225 condTerm ::= condFact (AND condFact)*
226 */
227 BloqueCondicion condTerm():{
228       BloqueCondicion c1, c2;
229 }{
230       c1=condFact()   (     <AND>   { System.out.println("label "+ c1.etqVerdad); }
231                             c2=condFact() {
232                                   System.out.println("label "+ c1.etqFalso);
233                                   System.out.println("\tgoto "+ c2.etqFalso);
234                                   c1 = c2;
235                                   }
236                 )*    { return c1; }
```

```
237 }
238 /*
239 condFact ::= (NOT)* ( condSimple | '[' cond ']' )
240 */
241 BloqueCondicion condFact():{
242       BloqueCondicion c1;
243       boolean negado = false;
244 }{    (<NOT> {negado = !negado;} )*
245       (     c1=condSimple()
246       |     <LCOR> c1=cond() <RCOR>
247       ) { if (negado) intercambiarCondicion(c1);
248           return c1;
249         }
250 }
251 /*
252 condSimple ::= expr (('>'|'<'|'='|'>='|'<='|'!=') expr)*
253 */
254 BloqueCondicion condSimple():{
255       String e1, e2;
256 }{
257       e1=expr()     (
258                     <MAYOR> e2=expr()  { return usarOpRelacional(e1, e2, ">"); }
259               |     <MENOR> e2=expr()  { return usarOpRelacional(e1, e2, "<"); }
260               |     <IGUAL> e2=expr()    { return usarOpRelacional(e1, e2, "="); }
261               |     <MAI> e2=expr()      { return usarOpRelacional(e1, e2, ">="); }
262               |     <MEI> e2=expr()      { return usarOpRelacional(e1, e2, "<="); }
263               |     <DIF> e2=expr()      { return usarOpRelacional(e1, e2, "!="); }
264               )
265 }
266 String id():{}{
267       <ID>      { return token.image; }
268 }
269 String numero():{}{
270       <NUMERO> { return usarOpAritmetico(token.image, "", ""); }
271 }
```

Las funciones de las líneas 21 a 50 permiten clarificar las acciones semánticas repartidas entre las reglas BNF. Por último, las etiquetas declaradas en la línea 127 podrían haberse fusionado en una sola, ya que nunca van a utilizarse dos de ellas simultáneamente; no se ha hecho así para mejorar la legibilidad del código.