Procesadores de Lenguajes

Ejercicios del Tema 7

Ejercicio 7.1

(a) ETDS

Asignación → **id** "**=**" *Expresion* "**;**"

{ Asignación.código = Expresión.código +
id + " = " + Expresión.temp + "\n"; }

Expresión → *Expresión* "+" *Término*

{ Expresión.temp = getNewTemp();
Expresión.código = Expresión1.código +
Término.codigo +
Expresión.temp +" = "+Expresión1.temp+ "+" +Término.temp+"\n";}

Expresión → *Expresión* "**-**" *Término*

{ Expresión.temp = getNewTemp();
Expresión.código = Expresión1.código +
Término.codigo +
Expresión.temp +" = "+Expresión1.temp+ "-" +Término.temp+"\n";}

Expresión → *Término*

{ Expresión.temp = Término.temp;
Expresión.código = Término.código;}

Expresión → "**-**" *Término*

{ Expresión.temp = getNewTemp();
Expresión.código = Término.codigo +
Expresión.temp + " = " + "-" Término.temp+"\n";}

Término → *Término* "*" *Factor*

{ Término.temp = getNewTemp();
Término.código = Término1.código +
Factor.codigo +
Término.temp + " = " + Término1.temp + "*" + Factor.temp+"\n";}

Término → *Término* "/" *Factor*

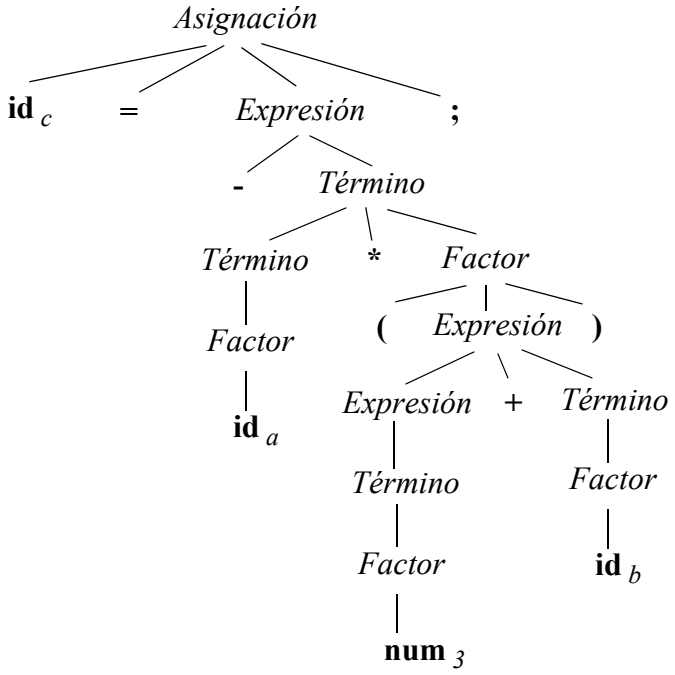
{ Término.temp = getNewTemp();
Término.código = Término1.código +
Factor.codigo +
Término.temp + " = " + Término1.temp + "/" + Factor.temp+"\n";}

Término → *Factor*

{ Término.temp = Factor.temp;
Término.código = Factor.código;}

```
Factor → num
        { Factor.temp = getNewTemp();
          Factor.código = Factor.temp + "=" + num.valor + "\n";}
Factor → id
        { Factor.temp = getPos(id);
          Factor.código = "";}
Factor → "(" Expresión ")"
        { Factor.temp = Expresión.temp;
          Factor.código = Expresión.código;}
```

(b) Árbol de análisis sintáctico:



Código intermedio:

temp0 = 3

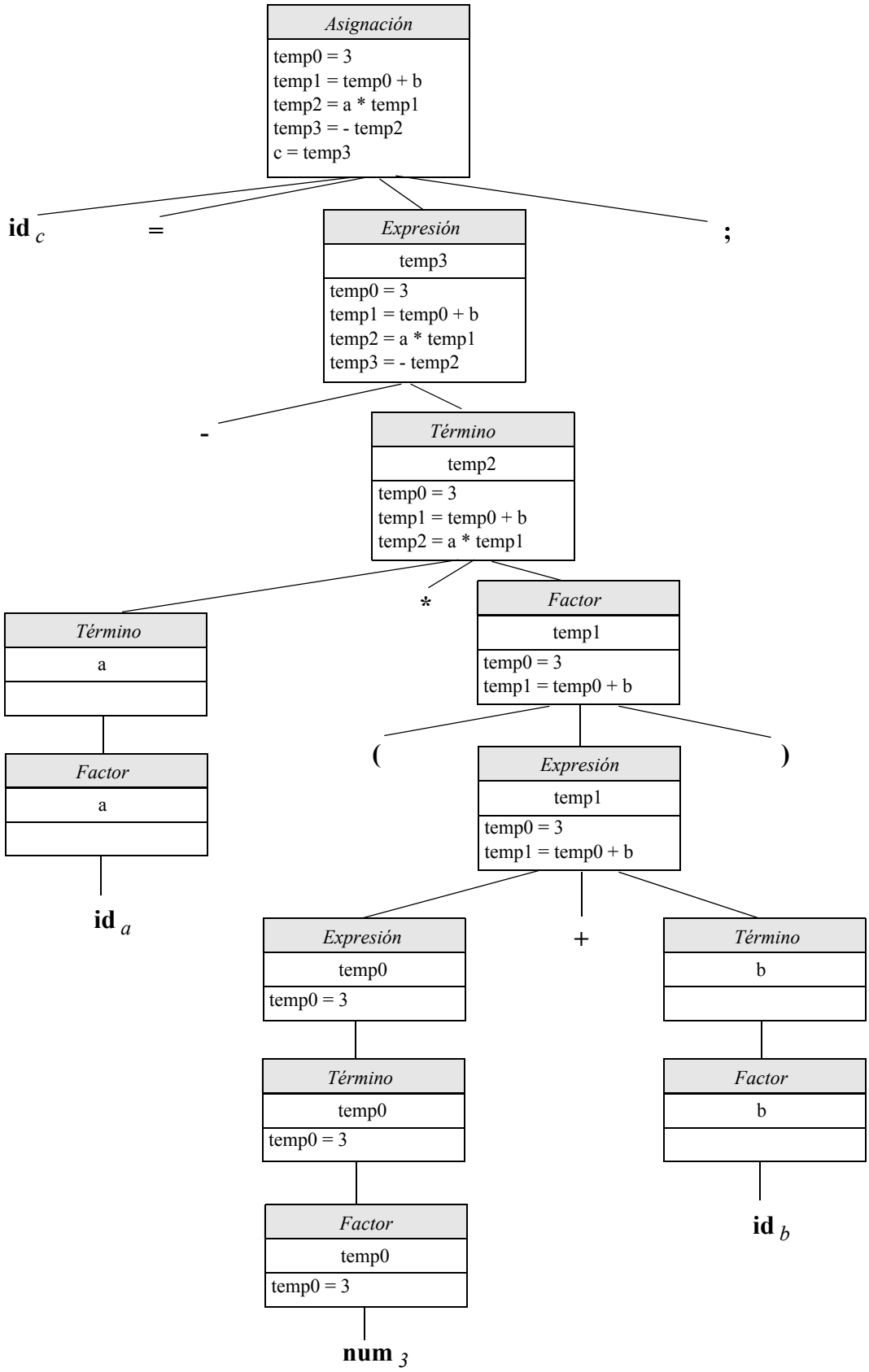
temp1 = temp0 + b

*temp2 = a * temp1*

temp3 = - temp2

c = temp3

A continuación se detalla el contenido de los atributos de los símbolos no terminales en el árbol de análisis sintáctico. El primer recuadro se refiere al atributo *temp* que contiene la referencia a la variable que almacena el valor del símbolo. El segundo recuadro contiene el valor del atributo *código*.


Asignación
temp0 = 3
temp1 = temp0 + b
temp2 = a * temp1
temp3 = - temp2
c = temp3
id c
=
Expresión
temp3
temp0 = 3
temp1 = temp0 + b
temp2 = a * temp1
temp3 = - temp2
;
-
Término
temp2
temp0 = 3
temp1 = temp0 + b
temp2 = a * temp1
Término
a
*
Factor
temp1
temp0 = 3
temp1 = temp0 + b
Factor
a
(
Expresión
temp1
temp0 = 3
temp1 = temp0 + b
)
id a
Expresión
temp0
temp0 = 3
+
Término
b
Término
temp0
temp0 = 3
Factor
b
Factor
temp0
temp0 = 3
id b
num 3

Ejercicio 7.2

Ejercicio 7.3

(a) Como vimos en el ejercicio 1, los atributos necesarios para generar el código de la instrucción de asignación son *temp* (referencia a la variable que almacena el valor de un símbolo) y *codigo* (código intermedio correspondiente a cada símbolo).

(b) El código de la clase es:

```
public class Simbolo {
  public String temp;
  public String codigo;

  public Simbolo() {
   this.temp = "";
   this.codigo = "";
  }
}
```

(c) La especificación en JavaCC del símbolo *Asignación* es la siguiente:

```
String Asignacion():
{
  Token id;
  Simbolo expr;
  String codigo;
}
{
  id = <ID>
  <ASIG>
  expr = Expresion()
  <PYC>
  {
    codigo = expr.codigo;
    codigo += getPos(id.image)+" = "+expr.temp+"\n";
    return codigo;
  }
}
```

La especificación del símbolo *Expresión* es la siguiente:

```
Simbolo Expresion():
{
  Simbolo term1;
  Simbolo term2;
  Simbolo expr;
  String newtemp;
  String op;
}
{
  term1 = Termino()
    {
      expr = new Simbolo();
      expr.temp = term1.temp;
      expr.codigo = term1.codigo;
    }
  (
   ( <MAS> { op = "+"; } | <MENOS> { op = "-"; })
   term2 = Termino()
    {
      expr.codigo += term2.codigo;
      newtemp = getNewTemp();
      expr.codigo += newtemp+"="+expr.temp+op+term2.codigo+"\n";
      expr.temp = newtemp;
    }
  )*
    {
      return expr;
    }
}
```

La especificación del símbolo *Término* es muy similar a la de *Expresión*:

```
Simbolo Termino():
{
  Simbolo fact1, fact2, term;
  String temp, op;
}
{
  fact1 = Factor()
    {
      term = new Simbolo();
      term.temp = fact1.temp;
      term.codigo = fact1.codigo;
    }
  (
   ( <PROD> { op = "*"; } | <DIV> { op = "/"; })
   fact2 = Factor()
    {
      newtemp = getNewTemp();
      term.codigo += fact2.codigo;
      term.codigo += newtemp+"="+term.temp+op+fact2.codigo+"\n";
      term.temp = newtemp;
    }
  )*
    { return term; }
}
```

La especificación del símbolo Factor es la siguiente:

```
Simbolo Factor():
{
  Simbolo fact,expr;
  Token num, id;
}
{
  num = <NUM>
    {
      fact = new Simbolo();
      fact.temp = getNewTemp();
      fact.codigo = fact.temp+ " = "+num.image+"\n";
      return fact;
    }
 | id = <ID>
    {
      fact = new Simbolo();
      fact.temp = getPos(id.image);
      fact.codigo = "";
      return fact;
    }
 | <PARAB> expr = Expresion() <PARCE>
    {
      return expr;
    }
}
```

Ejercicio 7.4

Ejercicio 7.5

Ejercicio 7.6

SOLUCIÓN:

(a) Los atributos asociados a cada símbolo van a ser los siguientes:

Símbolo	Atributos
Condición	**code**, **label_true**, **label_false**
CondiciónAnd	**code**, **label_true**, **label_false**
CondiciónBase	**code**, **label_true**, **label_false**
Operador	**code**
Expresión	**code**, **temp**

Para comprender el código a generar es bueno comenzar con un caso base. Por ejemplo, la condición "a = = b" se puede procesar mediante el siguiente código intermedio:

```
if a == b goto etiqueta1
goto etiqueta2
```

En este caso, el valor del atributo *label_true* de la condición deberá ser "*etiqueta1*" y el del atributo *label_false* será "*etiqueta2*". Estos valores deberán ser generados mediante el método getNewLabel() antes de crear el código. El código anterior significa que si se cumple la condición "a = = b" se realizará un salto a la etiqueta "*etiqueta1*", mientras que si la condición no se cumple el salto se realiza a la etiqueta "*etiqueta2*".

En un caso más general en el que la condición base esté formada por expresiones aritméticas más complejas, habrá que añadir el código que evalúe estas expresiones y añadir las dos instrucciones anteriores. La estructura del código resultante es la siguiente:

```
Código de la Expresión1
Código de la Expresión2
if Expresión1.temp Operador.code Expresión2.temp goto etiqueta1
goto etiqueta2
```

Para generar el código asociado a un operación AND entre dos condiciones hay que tener en cuenta que si alguna de las dos es falsa entonces el resultado es falso, mientras que si las dos son verdaderas el resultado es verdadero. Una posible estructura para el código resultante es la siguiente:

```
   Código del Operando1
Operando1.label_false:
   goto Operando2.label_false
Operando1.label_true:
   Código del Operando2
```

Las etiquetas verdadero y falso de la operación AND se toman como las etiquetas verdadero y falso del segundo operador. Si el primer operando es falso, se salta a la etiqueta falsa de la operación AND. Si el resultado del primer operador es verdadero se salta al código que evalúa al segundo operador.

El código asociado a las operaciones OR es parecido al anterior. En este caso hay que tener en cuenta que si alguno de los operandos es verdadero el resultado es verdadero, mientras que si ambos son falsos el resultado de la operación OR es falso. Una posible estructura para el código de la operación OR es la si-guiente:

```
   Código del Operando1
Operando1.label_true:
   goto Operando2.label_true
Operando1.label_false:
   Código del Operando2
```

En este caso, las etiquetas verdadero y falso de la operación OR se toman como las etiquetas verdadero y falso del segundo operador. Si el primer operando es verdadero, se salta a la etiqueta verdadera de la ope-ración OR. Si el resultado del primer operador es falso se salta al código que evalúa al segundo operador.

Con las consideraciones anteriores, el ETDS queda de la siguiente forma:

Condición → *Condición* "||" *CondiciónAnd*

{ Condición.label_true = CondiciónAnd.label_true;
  Condición.label_false = CondiciónAnd.label_false;
  Condición.code = Condición1.code +
  Condición1.label_true+ ":"+ "\n"+
  " goto "+CondiciónAnd.label_true+ "\n" +
  Condición1.label_false+ ":"+ "\n"+
  CondiciónAnd.code; }

Condición → *CondiciónAnd*
{ Condición.code = CondiciónAnd.code;
Condición.label_true = CondiciónAnd.label_true;
Condición.label_false = CondiciónAnd.label_false; }

CondiciónAnd → *CondiciónAnd* "**&&**" *CondiciónBase*
{ CondiciónAnd.label_true = CondiciónBase.label_true;
CondiciónAnd.label_false = CondiciónBase.label_false;
CondiciónAnd.code = CondiciónAnd1.code +
CondiciónAnd1.label_false+ ":"+ "\n"+
" goto "+CondiciónBase.label_false+ "\n" +
CondiciónAnd1.label_true+ ":"+ "\n"+
CondiciónBase.code; }

CondiciónAnd → *CondiciónBase*
{ CondiciónAnd.code = CondiciónBase.code;
CondiciónAnd.label_true = CondiciónBase.label_true;
CondiciónAnd.label_false = CondiciónBase.label_false; }

CondiciónBase → *Expresión Operador Expresión*
{ CondiciónBase.label_true = getNewLabel();
CondiciónBase.label_false = getNewLabel();
CondiciónBase.code = Expresión1.code +
Expresión2.code +
"if "+Expresión1.temp+Operador.code+ Expresión2.temp+
" goto "+CondiciónBase.label_true+ "\n";
"goto "+CondiciónBase.label_false+ "\n"; }

CondiciónBase → "(" *Condición* ")"
{ CondiciónBase.code = Condición.code;
CondiciónBase.label_true = Condición.label_true;
CondiciónBase.label_false = Condición.label_false; }

Operador → "= =" { Operador.code = "= ="; }

Operador → "**!**=" { Operador.code = "!="; }

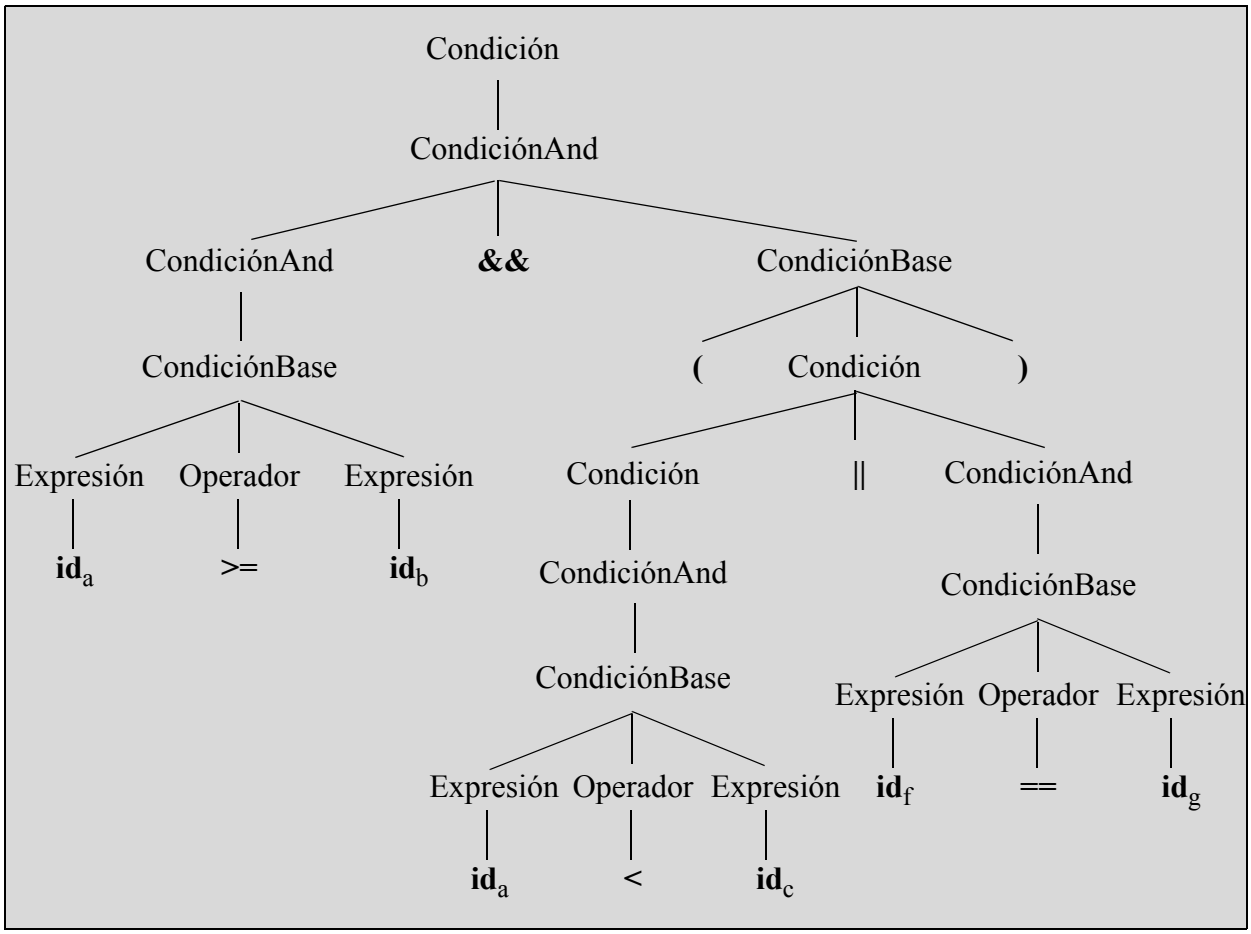
Operador → ">" { Operador.code = ">"; }

Operador → "<" { Operador.code = "<"; }

Operador → ">=" { Operador.code = ">="; }

Operador → "<=" { Operador.code = "<="; }

(b) Árbol de análisis sintáctico para la expresión condicional "a >= b && (a<c || f ==g)"



Código intermedio:

```
  if a >= b goto etiqueta1
  goto etiqueta2
etiqueta2:
  goto etiqueta6
etiqueta1:
  if a < c goto etiqueta3
  goto etiqueta4
etiqueta3:
  goto etiqueta5
etiqueta4:
  if f == g goto etiqueta5
  goto etiqueta6
```

Los valores de las etiquetas son:

- label_true = etiqueta5
- label_false = etiqueta6

Ejercicio 7.7

Ejercicio 7.8

Ejercicio 7.9

Solución

```
Inst InstIf():
{
  Condition cond;
  Inst inst_then, inst_else, inst_if;
  boolean else_flag = false;
}
{
  <IF> <PARAB> cond = Condicion() <PARCE>
  inst_then = Instruccion()
  [ <ELSE> inst_else = Instruccion() { else_flag = true; } ]
    {
      inst_if = new Inst();
      inst_if.code = cond.code;
      inst_if.code += cond.label_true+":";
      inst_if.code += inst_then.code;
      String newlabel = getNewLabel();
      if(else_flag) inst_if += "goto "+newlabel+"\n";
      inst_if.code += cond.label_false+":";
      if(else_flag) {
        inst_if.code += inst_else.code;
        inst_if.code += newlabel+":";
      }
      return inst_if;
    }
}
```

Ejercicio 7.10

La estructura del código de la instrucción *for* es la siguiente:

```
  Código de la instrucción de inicio
etiqueta_inicio:
  Código de la condición
etiqueta_condición_verdadera:
  Código de la instrucción del bucle
  Código de la instrucción de incremento
  goto etiqueta_inicio
etiqueta_condición falsa:
```

Esquema de traducción basado en el formato de la herramienta JavaCC.:

```
Inst InstFor() :
{
  Inst inst1, inst2, inst3;
  Inst instfor;
  Condition cond;
}
{
  <FOR> <PARAB>
  inst1 = Instruccion() <PYC>
  cond = Condicion() <PYC>
  inst2 = Instruccion() <PARCE>
  inst3 = Instruccion()
    {
      String etiqueta_inicio = getNewLabel();
      instfor = new Inst();
      instfor.code = inst1.code;
      instfor.code += etiqueta_inicio+ ":"+ "\n";
      instfor.code += cond.code;
      instfor.code += cond.label_true+ ":"+ "\n";
      instfor.code += inst3.code;
      instfor.code += inst2.code;
      instfor.code += "  goto "+etiqueta_inicio+ "\n";
      instfor.code += cond.label_false+ ":"+ "\n";
      return instfor;
    }
}
```

Ejercicio 7.11

Ejercicio 7.12

El esquema del código a generar es el siguiente:

```
label_false:
Instruccion.code
Instruccion.code
...
Instruccion.code
Condicion.code
label_true:
```

La única dificultad que impone este esquema es que no se conoce el valor de *label_false* hasta que no se reconoce el símbolo *Condicion*. Por tanto, es necesario almacenar el código de las instrucciones en una variable intermedia (*temp*) y construir el valor de *code* después de reconocer *Condicion*. La solución, en el formato de la herramienta JavaCC es la siguiente:

```
Inst InstRepeat() :
{
  Inst inst_repeat = new Inst();
  Inst inst;
  String temp = "";
  Condition cond;
}
{
  <REPEAT>
  <LLAVEAB>
  ( inst = Instruccion() { temp += inst.code; } )*
  <LLAVECE>
  <UNTIL>
  <PARAB>
  cond = Condicion()
  <PARCE>
  <PYC>
  { inst_repeat.code = cond.label_false+":"+ "\n";
    inst_repeat.code += temp;
    inst_repeat.code += cond.code;
    inst_repeat.code += cond.label_true+":"+ "\n";
    return inst_repeat;
  }
}
```

Ejercicio 7.13

Ejercicio 7.14

Ejercicio 7.15