Procesadores de lenguaje

Ejercicios del Tema 5

Ejercicio 5.1

La siguiente figura muestra una gramática LL(1) que reconoce una lista de números separados por coma.

Lista → **num** *SigueLista*

SigueLista → **coma num** *SigueLista*

SigueLista → λ

Partiendo de esta gramática, desarrolle un ETDS que genere la estructura de datos asociada a la lista. Para representar la lista se utilizará objetos de la clase *ListElement* que permiten describir un elemento de la lista. El campo *next* debe apuntar al siguiente elemento de la lista, de manera que la lista completa se puede representar como la referencia al primer elemento de la lista.

```
public class ListElement {

  public double value;
  public ListElement next;

  public ListElement(double val) {
    this.value = val;
    this.next = null;
  }
}
```

Ejercicio 5.2

La siguiente figura muestra una gramática LL(1) que describe el operador potencia.

Factor → **num** *Potencia*

Potencia → **elev** **num** *Potencia*

Potencia → λ

La potencia es un operador binario que tiene la particularidad de considerarse asociativo a la derecha. Esto quiere decir que una expresión del tipo

a ^ b ^ c

debe entenderse como

a ^ (b ^ c)

al contrario de lo que ocurre con la mayoría de operadores binarios como la suma, el producto, etcétera.

Considere las siguientes clases que permiten definir las potencias de números:

```
// Clase abstracta que describe una expresión aritmética
public abstract class  Expression {
}

// Clase que describe un número constante
public class Number extends Expression {
  private double value;

  public Number(double val) {  this.value = val; }
}

// Clase que describe la potencia entre dos expresiones
public class Power extends Expression {
  private Expression base;
  private Expression pow;

  public Power(Expression a, Expression b) {
    this.base = a;
    this.pow = b;
  }
}
```

Desarrolle un ETDS que genere el árbol de sintaxis abstracta asociado a una expresión formada por el operador potencia.

Ejercicio 5.3

La siguiente figura muestra la descripción de una lista de elementos en el formato de JavaCC.

```
void List() :
{}
{
   Element()  (  <COMMA>  Element()  )*
}
```

Se supone que el símbolo *Element()* devuelve un objeto de la clase *DoubleListElement*. Esta clase permite desarrollar una lista doblemente enlazada, donde el campo *prev* de cada elemento contiene el enlace al elemento anterior y el campo *next* contiene el enlace al elemento posterior. El objeto *DoubleListElement* que devuelve el símbolo *Element()* tiene estos enlaces a nulo.

```
public class DoubleListElement {

  public double value;
  public DoubleListElement prev;
  public DoubleListElement next;

  public DoubleListElement(double val) {
    this.value = val;
    this.prev = null;
    this.next = null;
  }
}
```

Modifique la descripción de JavaCC para que el símbolo *List()* devuelva una lista doblemente enlazada formada por los elementos reconocidos.

Ejercicio 5.4

La siguiente gramática describe la sintaxis de las expresiones regulares:

Expr → *Option*

Expr → *Expr* **or** *Option*

Option → *Base*

Option → *Option* *Base*

Base → **symbol**

Base → **lparen** *Expr* **rparen** *Oper*

Oper → λ

Oper → **star**

Oper → **plus**

Oper → **hook**

Desarrolle a partir de esta gramática un ETDS que genere el árbol de sintaxis abstracta que describa la expresión regular. La estructura de datos a utilizar se describe en la próxima página.

```
// Clase abstracta que describe una expresión regular
public abstract class  Expression {
}

// Clase que describe un símbolo del lenguaje
public class Symbol extends Expression {
  private char symbol;

  public Symbol(char s) {  this.symbol = s; }
}

// Clase que describe un lista de opciones " a |  b  | c "
public class OptionList extends Expression {
  private Expression[] list;

  public OptionList(Expression exp) {
    this.list = new Expression[1];
    this.list[0] = exp;
  }
  public void addOption(Expression exp) { ... }
}

// Clase que describe una concatenación de expresiones " a  b  c "
public class ConcatList extends Expression {
  private Expression[] list;

  public ConcatList(Expression exp) {
    this.list = new Expression[1];
    this.list[0] = exp;
  }
  public void concat(Expression exp) { ... }
}

// Clase que describe una operación de clausura " ( a )* ",
// clausura positiva " ( a )+ ", o una opcionalidad " ( a )? "
 public class Operation extends Expression {
  public static final int STAR = 1;
  public static final int PLUS = 2;
  public static final int HOOK = 3;

  private int operator;
  private Expression operand;

  public Operation(int op, Expression exp) {
    this.operartor = op;
    this.operand = exp;
  }
}
```

Ejercicio 5.5

La siguiente gramática describe la sintaxis de las expresiones regulares en el formato de la herramienta JavaCC:

```
void Expr() :
{}
{
   Option() ( <OR>   Option() )*
}

void Option() :
{}
{
   (  Base()  )+
}

void Base() :
{}
{
  <SYMBOL>
 |<LPAREN>  Expr() <RPAREN> Oper()
}

void Oper():
{}
{
 ( <STAR>  |  <PLUS>  |  <HOOK>  )?
}
```

Modifique la descripción de JavaCC para que el símbolo *Expr()* devuelva la definición de la expresión regular utilizando la estructura de datos de la página anterior.

Ejercicio 5.6

La siguiente gramática permite definir una expresión aritmética en notación prefija:

Expr → **num**

Expr → **op lparen** *Expr* **comma** *Expr* **rparen**

A continuación se muestra una expresión en el formato planteado:

+(*(3, 5) , /(-(8, 1), 6))

Partiendo de esta gramática, desarrolle un ETDS que genere la estructura de datos asociada a la expresión. Para representar las expresiones se utilizarán objetos de la clase abstracta *Expression*. Para representar los números se utiliza la clase *Number* mientras que para representar las operaciones se utiliza la clase *Operation*. Ambas son subclases de *Expression*.

```
public abstract class Expression {
}

public class Number extends Expression {
  private double number;

  public Number(double number) { ... }
}

public class Operation extends Expression {
  private int op;
  private Expression left;
  private Expression right;

  public Operation(int op, Expression left, Expresion right) { ... }
}
```

Ejercicio 5.7

Un clasificador es un sistema que toma como entrada un patrón, que contiene un conjunto de valores para una serie de atributos, y genera como salida la clase a la que pertenece dicho patrón. Una de las representaciones más utilizadas para el desarrollo de clasificadores son los árboles de decisión. En un árbol de decisión, los nodos no terminales se refieren a preguntas sobre el valor de un cierto atributo del patrón. Las ramas del nodo se refieren a los posibles valores del atributo. Los nodos terminales del árbol representan el valor de la clase seleccionada por el clasificador para el patrón dado.

La gramática presentada a continuación permite describir árboles de decisión:

Árbol → **tree id** "**{**" *Nodo* "**}**"
Nodo → *NodoNoTerminal*
Nodo → *NodoTerminal*
NodoNoTerminal → **switch** *Atributo* "**{**" *ListaDeCasos* "**}**"
NodoTerminal → **class** *Clase* "**;**"
ListaDeCasos → *Caso ListaDeCasos*
ListaDeCasos → λ
Caso → **case** *Valor* "**:**" *Nodo*
Atributo → **id**
Clase → **id**
Valor → **num**

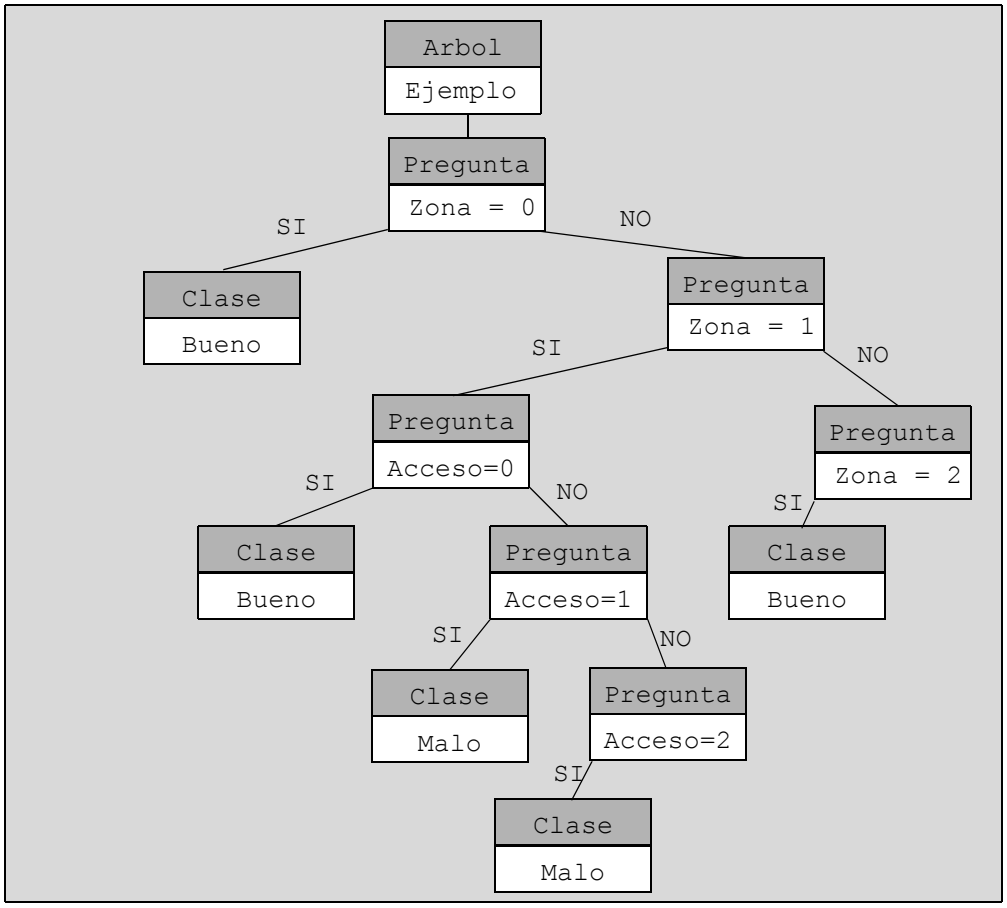
La siguiente figura muestra un ejemplo de descripción de un árbol de decisión con patrones basados en dos atributos (*Zona* y *Acceso*) y dos clases a seleccionar (*Bueno* y *Malo*).

```
tree Ejemplo {
   switch Zona {
      case 0: class Bueno;
      case 1: switch Acceso {
                 case 0: class Bueno;
                 case 1: class Malo;
                 case 2: class Malo;
              }
      case 3: class Bueno;
   }
}
```

Construya un ETDS que permita generar una estructura de datos (un objeto de la clase Arbol) a partir de la gramática planteada. Dicha estructura de datos se construirá por medio de las siguientes clases:

Nota: La gramática propuesta permite describir nodos con un número indefinido de ramas. Sin embargo, la estructura de datos propuesta sólo admite nodos binarios. El ejemplo presentado anteriormente debería generar una estructura de datos como la siguiente:

```
public abstract class Nodo {
}

public class Arbol {
  private String nombre;
  private Nodo nodo;

  public Arbol(String nombre, Nodo nodo) {
    this.nombre = nombre;
    this.nodo = nodo;
  }
}

public class Pregunta extends Nodo {
   private String atributo;
   private int valor;
   private Nodo cierto;
   private Nodo falso;

   public Pregunta(String atributo, int valor) {
     this.atributo = atributo;
     this.valor = valor;
     this.cierto = null;
     this.falso = null;
   }

   public void asignaCierto(Nodo cierto) {
     this.cierto = cierto;
   }

   public void asignaFalso(Nodo falso) {
      this.falso = falso;
   }
}

public class Clase extends Nodo {
   private String clase;

   public Clase(String clase) {
     this.clase = clase;
   }
}
```


Arbol
Ejemplo
Pregunta
Zona = 0
SI
NO
Clase
Bueno
Pregunta
Zona = 1
SI
NO
Pregunta
Acceso=0
Pregunta
Zona = 2
SI
NO
SI
Clase
Bueno
Pregunta
Acceso=1
Clase
Bueno
SI
NO
Clase
Malo
Pregunta
Acceso=2
SI
Clase
Malo

Ejercicio 5.8

La siguiente gramática permite describir un circuito formado por resistencias unidas en serie o en paralelo:

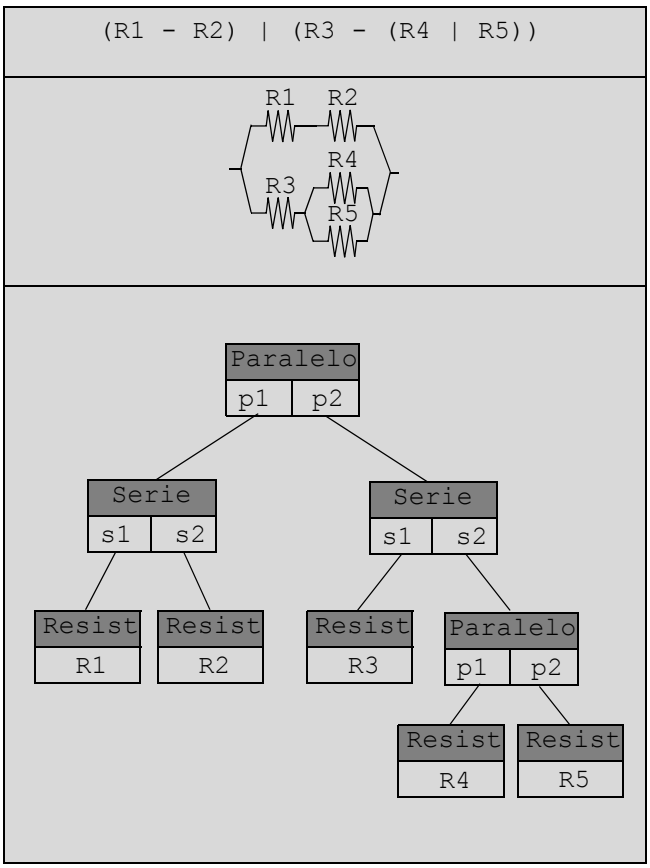
Circuito → *CircuitoSerie RamaParalela*
RamaParalela → "|" *CircuitoSerie RamaParalela*
RamaParalela → λ
CircuitoSerie → *CircuitoBase ConexionSerie*
ConexionSerie → "**-**" *CircuitoBase ConexionSerie*
ConexionSerie → λ
CircuitoBase → **resistencia**
CircuitoBase → "**(**" *Circuito* "**)**"

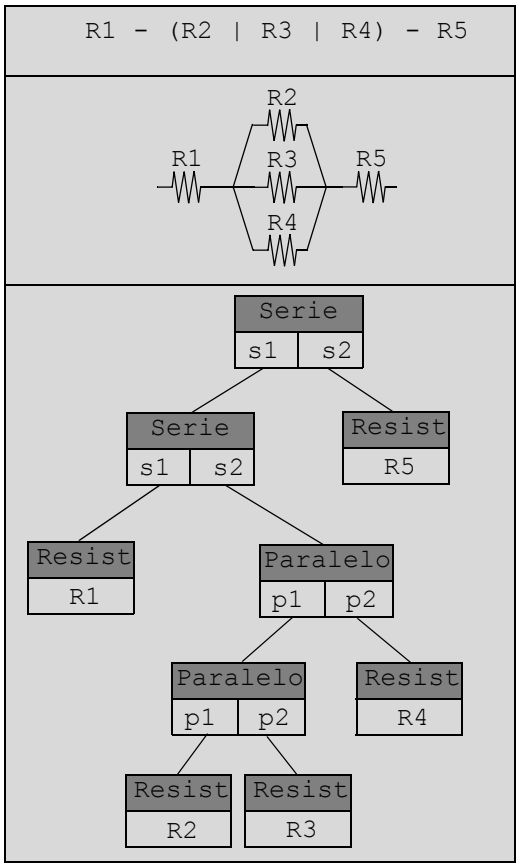
Considere que las siguientes clases están definidas:

```
public class Nodo {
}

public class Resistencia extends Nodo {
  public String R;
}

public class Serie extends Nodo {
  public Nodo s1;
  public Nodo s2;
}

public class Paralelo extends Nodo {
  public Nodo p1;
  public Nodo p2;
}
```

Desarrolle un ETDS que genere un árbol basado en estas estructuras a partir de una descripción en modo texto de un circuito.

A continuación se muestran dos ejemplos de circuitos, su representación textual y el árbol que debe generar el ETDS.





Ejercicio 5.9

La siguiente gramática permite describir una interfaz gráfica formada por un panel en el que se colocan una serie de componentes. El panel puede ser de tipo *Xbox*, que coloca los componentes de izquierda a derecha, o de tipo *YBox*, que coloca los componentes de arriba hacia abajo. A su vez, los componentes pueden ser etiquetas, campos, botones o nuevos paneles.

Panel → *XBox*

Panel → *YBox*

XBox → **xbox id** "**{**" *ListaComponentes* "**}**"

YBox → **ybox id** "**{**" *ListaComponentes* "**}**"

ListaComponentes → *Componente ListaComponentes*

ListaComponentes → λ

Componente → *Panel*

Componente → *Etiqueta*

Componente → *Campo*

Componente → *Botón*

Etiqueta → **label id** "**;**"

Campo → **field id** "**;**"

Botón → **button id** "**;**"

La siguiente figura muestra un ejemplo de descripción de un panel siguiendo el formato anterior, así como la representación gráfica del panel descrito.

```
ybox panel {
 label titulo;
 xbox box1 {
   ybox box2 {
     label etiqueta1;
     label etiqueta2;
     label etiqueta3;
   }
   ybox box3 {
     field campo1;
     field campo2;
     field campo3;
   }
 }
 button aceptar;
}
```

titulo	
etiqueta1	campo1
etiqueta2	campo1
etiqueta3	campo1
aceptar	

Considere que las siguientes clases están definidas:

```
public class Component {
}

public class XBox extends Component {
  public XBox(String id) { ... }
  public void add(Component comp) { ... }
}

public class YBox extends Component {
  public YBox(String id) { ... }
  public void add(Component comp) { ... }
}

public class Label extends Component {
  public Label(String id) { ... }
}

public class Field extends Component {
  public Field(String id) { ... }
}

public class Button extends Component {
  public Button(String id) { ... }
}
```

Desarrolle un ETDS que genere la estructura del panel basada en las clases anteriores.

Ejercicio 5.10

La gramática presentada a continuación permite describir una representación gráfica bidimensional por medio de particiones horizontales y verticales.

Figura → **figura** *Dimensión Partición*

Dimensión → "**[**" **entero** "**,**" **entero** "**]**"

Partición → *Horizontal*

Partición → *Vertical*

Partición → *Color*

Horizontal → **horizontal** *Factor* "**{**" *Partición* "**,**" *Partición* "**}**"

Vertical → **vertical** *Factor* "**{**" *Partición* "**,**" *Partición* "**}**"

Color → **color** "**(**" **entero** "**,**" **entero** "**,**" **entero** "**)**"

Factor → "**[**" **real** "**]**"

La dimensión contiene dos números enteros que expresan en pixels la anchura y altura de la figura descrita. El factor contiene un número real entre 0 y 1 que expresa el tamaño de la división como tanto por uno (p.e., el factor 0.3 representa una división del espacio del 30%). El color contiene la representación RGB de relleno del espacio resultante. Por ejemplo, la siguiente descripción representa un cuadrado rojo de 20x20 pixels:

```
figura [20,20] color (256,0,0)
```

Otro ejemplo: un rectángulo de 30x10 pixels dividido horizontalmente en dos trozos. El primer trozo mide 12x10 pixels y es de color azul. El segundo mide 18x10 pixels y es de color verde.

```
figura [30,10] horizontal [0.4] {
   color (0,0,256),
   color (0,256,0)
}
```

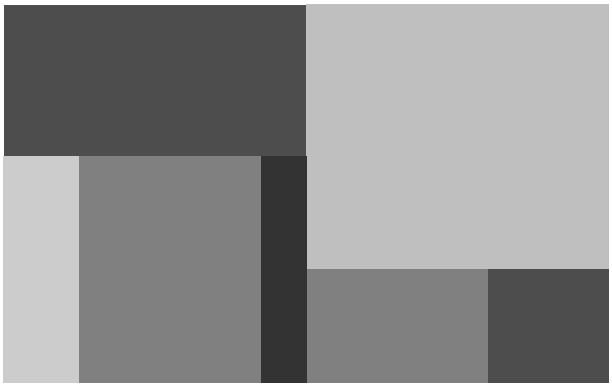
Construya un ETDS que permita generar una estructura de datos a partir de la gramática planteada. Dicha estructura de datos se construirá por medio de las siguientes clases:

```
public class Figura {
  int width;
  int height;
  Particion desc;

  Figura(int w, int h, Particion p) {
    this.width = w;
    this.height = h;
    this.desc = p;
  }
}
```

```
public abstract class Particion {
}

public class Horizontal extends Particion {
  Particion left, right;

  public Horizontal(Particion l, Particion r) {
     this.left = l;
     this.right = r;
  }
}

public class Vertical extends Particion {
  Particion top, bottom;

  public Vertical(Particion t, Particion b) {
     this.top = t;
     this.bottom = b;
  }
}

public class Rectangulo extends Particion {
  int width, height;
  int red, green, blue;

  public Rectangulo(int w, int h, int r, int g, int b) {
    this.width = w; this.height = h;
    this.red = r; this.green = g; this.blue = b;
  }
}
```

La siguiente figura muestra un ejemplo más complejo de descripción de una representación gráfica en el formato planteado.

```
figura [80,50] horizontal  [0.5] {
    vertical [0.4] {
      color (180,180,180) ,
      horizontal [0.25] {
         color (51,51,51),
         horizontal [0.8] {
            color (128,128,128),
            color (205,205,205)
         }
      }
    },
    vertical [0.7] {
      color (64,64,64) ,
      horizontal [0.6] {
        color (128,128,128),
        color (180,180,180)
      }
    }
}
```



Ejercicio 5.11

La siguiente gramática permite describir un dibujo formado por una serie de figuras. Cada escena contiene una lista de puntos (a los que se asigna un nombre) y una lista de figuras, definidas por medio de estos puntos (a los que se identifica por su nombre). Las figuras pueden ser líneas, rectángulos y polígonos.

Escena → **scene llaveab** *ListaDePuntos ListaDeFiguras* **llavece**

ListaDePuntos → *Punto ListaDePuntos*

ListaDePuntos → λ

Punto → **point id parab num coma num parce pyc**

ListaDeFiguras → *Figura ListaDeFiguras*

ListaDeFiguras → λ

Figura → *Línea*

Figura → *Rectángulo*

Figura → *Polígono*

Línea → **line parab id coma id parce pyc**

Rectángulo → **rectangle parab id coma id parce pyc**

Polígono → **polygon parab id** *ContinúaListaDePuntos* **parce pyc**

ContinúaListaDePuntos → **coma id** *ContinúaListaDePuntos*

ContinúaListaDePuntos → λ

Para representar la información de una escena se han definido las siguientes clases:

```
public class Scene {
  private Point[] point;
  private Figure[] figure;
  public Scene() { }
  public void addPoint( Point  p) { ... }
  public void addFigure( Figure f ) { ... }
  public Point searchPoint( String name) { ... }
}

public class Point {
  private String name;
  private double x,y;
  public Point(String name, double x, double y) { ... }
  public boolean equals(String name) { ... }
}

public abstract class Figure {
}
```

```
public class Line extends Figure {
  private Point p1,p2;
  public Line(Point p1, Point p2) { ... }
}

public class Rectangle extends Figure {
  private Point p1,p2;
  public Rectangle(Point p1, Point p2) { ... }
}

public class Polygon extends Figure {
  private Point[] plist;
  public Polygon(Point p) { ... }
  public void addPoint(Point p) { ... }
}
```

A continuación se presenta un ejemplo de especificación de una escena en este formato:

```
scene {
  point a( 1.0, 1.0 );
  point b( 1.0, 2.0 );
  point c( 3.2, 1.5 );
  point d( 2.4, 3.6 );

  line( a, c );
  rectangle( b, d );
  polygon( a, b, c );
}
```

Construya un ETDS que permita crear un objeto *Scene* como resultado del análisis de una especificación en el formato descrito. Enumere los atributos (heredados y sintetizados) asignados a cada símbolo.

Ejercicio 5.12

La siguiente gramática SLR representa la sintaxis de la instrucción de asignación de un lenguaje basado en conjuntos:

Asig → **id igual** *Expr* **pyc**

Expr → *Comp*

Expr → *Expr* **union** *Comp*

Expr → *Expr* **interseccion** *Comp*

Comp → **complemento** *Base*

Comp → *Base*

Base → **llave_ab** *Lista* **llave_ce**

Base → **par_ab** *Expr* **par_ce**

Base → **id**

Lista → λ

Lista → *Elem*

Elem → **num**

Elem → *Elem* **coma num**

A continuación se muestra un ejemplo de una cadena de entrada para esta gramática:

A = { 1 , 2 } ∪ ({ 3 , 4, 5 } ∩ { 3 , 5 }) ∪ ¬ B ;

Con el fin de describir internamente las instrucciones de asignación, se han desarrollado las siguientes clases:

```
public class AssignInst {
  private Variable var;
  private Set value;

  public AssignInst(Variable var, Set value) {
    this.var = var;
    this.value = value;
  }
}

public abstract class Set {
  public abstract boolean contains(int num);
}
```

```
public class Variable extends Set {
  private String name;
  private Set value;

  public boolean contains(int num) {
   return this.value.contains(num);
  }
}

public class EnumSet extends Set {
  private int[] elem;

  public EnumSet() { this.elem = new int[0]; }
  public void addElement(int num) { ... }
  public boolean contains(int num) { ... }
}

public class Complement extends Set {
  private Set operand;

 public Complement(Set operand) { this.operand=operand; }
  public boolean contains(int num) {
    return !operand.contains(num);
  }
}

public class Union extends Set {
  private Set left;
  private Set right;

  public Union(Set left, Set right) {
    this.left = left;
    this.right = right;
  }

  public boolean contains(int num) {
    return left.contains(num) || right.contains(num);
  }
}

public class Intersection extends Set {
  private Set left;
  private Set right;

  public Intersection(Set left, Set right) {
    this.left = left;
    this.right = right;
  }

  public boolean contains(int num) {
    return left.contains(num) && right.contains(num);
  }
}
```

Se dispone además del método *getVariable(String id)* que obtiene un objeto *Variable* a partir de su nombre.

Construya un ETDS sobre la gramática que permita crear un objeto *AsigInst* como resultado del análisis. Enumere los atributos (heredados y sintetizados) asignados a cada símbolo.

Ejercicio 5.13

A continuación se muestra una gramática LL(1) para la instrucción de asignación de conjuntos. Utilizando las clases descritas en el ejercicio anterior, construya un ETDS sobre la gramática LL1) que permita crear un objeto *AsigInst* como resultado del análisis. Enumere los atributos (heredados y sintetizados) asignados a cada símbolo.

Asig → **id** **igual** *Expr* **pyc**

Expr → *Comp* *SigueExpr*

SigueExpr → **union** *Comp* *SigueExpr*

SigueExpr → **interseccion** *Comp* *SigueExpr*

SigueExpr → λ

Comp → **complemento** *Base*

Comp → *Base*

Base → **llave_ab** *Lista* **llave_ce**

Base → **par_ab** *Expr* **par_ce**

Base → **id**

Lista → λ

Lista → **num** *SigueLista*

SigueLista → λ

SigueLista → **coma** **num** *SigueLista*

Ejercicio 5.14

La siguiente figura muestra la descripción de la instrucción de asignación de conjuntos en el formato EBNF de la herramienta JavaCC. Modifique esta descripción para que el símbolo *Asig()* devuelva un objeto de tipo *AsigInst*.

```
void Asig() :
{}
{
  <ID> <IGUAL> Expr() <PYC>
}

void Expr() :
{}
{
  Comp()  ( (<UNION> | <INTERSECCION>)  Comp() )*
}

void Comp() :
{}
{
  ( <COMPLEMENTO> )?  Base()
}

void Base() :
{}
{
    <ID>
  | <LLAVEAB> ( <NUM> ( <COMA> <NUM> )* )? <LLAVECE>
  | <PARAB> Expr() <PARCE>
}
```

Ejercicio 5.15

La siguiente gramática permite describir un circuito formado por resistencias unidas en serie o en paralelo:

Circuito → *Circuito* **paralelo** *CircuitoSerie*

Circuito → *CircuitoSerie*

CircuitoSerie → *CircuitoSerie* **serie** *CircuitoBase*

CircuitoSerie → *CircuitoBase*

CircuitoBase → **resistencia**

CircuitoBase → **parab** *Circuito* **parce**

Considere que las siguientes clases están definidas:

```
public abstract class Nodo {
}

public class Resistencia extends Nodo {
  public String R;
}

public class Serie extends Nodo {
  public Nodo s1;
  public Nodo s2;
}

public class Paralelo extends Nodo {
  public Nodo p1;
  public Nodo p2;
}
```

A continuación se muestran dos ejemplos de descripción de circuitos reconocidos por la gramática propuesta. El carácter "|" representa el token paralelo. Por su parte, el token serie se representa por medio del carácter "**-**".

```
(R1 - R2) | (R3 - (R4 | R5))
```

```
R1 - (R2 | R3 | R4) - R5
```

Desarrolle un ETDS que genere una estructura de datos (una referencia a un objeto de la clase *Nodo*) que contenga la descripción de un circuito a partir de su definición en modo texto.

Ejercicio 5.16

En general, la parte derecha de una regla en Prolog (el antecedente) está formada por una lista de términos separados por coma (la coma indica una operación lógica AND). Estos términos pueden ser, a su vez, una expresión relacional (por ejemplo, X = Y), un corte (!) o una disyunción. Las disyunciones se definen como una lista de términos separados por ';' (el punto y coma indica una operación lógica OR). Las expresiones permiten definir las operaciones de unificación y de comparación. Existen varios operadores de unificación (por ejemplo, '=', '==', '\=', 'is', '=\=', '=:=') y de comparación ('>', '<', '=<', '>=') pero las vamos a englobar en un único token llamado **relop**. Los operandos de una relación pueden ser expresiones aritméticas, variables, constantes numéricas, predicados y listas. La siguiente gramática describe en notación BNF la sintaxis comentada.

Antecedente → *Conjunción*

Conjunción → *Término* *SigueConjunción*

SigueConjunción → **comma** *Término* *SigueConjunción*

SigueConjunción → λ

Término → **lparen** *Disyunción* **rparen**

Término → **cut**

Término → *Relación*

Disyunción → *Conjunción* *SigueDisyunción*

SigueDisyunción → **semicolon** *Conjunción* *SigueDisyunción*

SigueConjunción → λ

Relación → *Expresión* *SigueRelación*

SigueRelación → **relop** *Expresión*

SigueRelación → λ

Se desea enriquecer la gramática anterior para realizar un ETDS que genere la estructura de datos del antecedente de las reglas. Para ello se consideran las siguientes clases: *Term,* (clase abstracta que engloba a todas las formas que puede tomar un término); *AndList*, (clase que representa la conjunción de una lista de términos); *OrList*, (clase que representa la disyunción de una lista de términos); *CutTerm*, (clase que representa el operador de corte); *Relation*, (clase que representa una relación entre dos expresiones); y *Expression*, (clase que representa el operando de una relación). Se supone que el símbolo *Expresión* posee un atributo sintetizado que almacena el objeto *Expression* con la información de la expresión correspondiente. A continuación se muestra el código de estas clases:

```
public abstract class Term {
}

public class AndList extends Term {
  public Term term;
  public AndList next;
}

public class OrList extends Term {
  public Term term;
  public OrList next;
}

public class CutTerm extends Literal {
  public CutTerm() { }
}

public class Relation extends Term {
  public Expression left;
  public Expression right;
  public int operator;
}

public class Expression extends Term {
  ...
}
```

Ejercicio 5.17

Una *taxonomía* permite definir de forma jerárquica los diferentes grupos en los que se pueden clasificar los elementos de un conjunto. Cada grupo se denomina un *taxón* y cada nivel de profundidad jerárquica se denomina una *categoría taxonómica*.

La siguiente gramática describe la sintaxis de lenguaje de definición de taxonomías :

Taxonomía → **taxonomy id lbrace** *ListaDeTaxones* **rbrace**

ListaDeTaxones → λ

ListaDeTaxones → *Taxón SigueLista*

SigueLista → **comma** *Taxón SigueLista*

SigueLista → λ

Taxón → *Individuo*

Taxón → *Categoría*

Individuo → **lbracket id rbracket**

Categoría → **id lbrace** *ListaDeTaxones* **rbrace**

A continuación se muestra un ejemplo de clasificación taxonómica:

```
taxonomy Selección {
  Portería { [Casillas], [Palop], [Reina] },
  Defensa { Central { [Marchena],[Puyol],[Juanito],[Albiol] },
            LateralIzquierdo { [Capdevila] },
            LateralDerecho { [Ramos],[Navarro],[Arbeloa] } },
  Media { Centrocampista { [Xavi],[Iniesta],[Senna],[Cesc],[DeLaRed],[Alonso] },
          ExtremoIzquierdo { [Silva] },
          ExtremoDerecho { [Cazorla] } },
  Delantera { [Villa], [Torres], [Guiza], [García] }
}
```

La forma más sencilla de representar una taxonomía es un árbol donde la raíz sería el nombre de la taxonomía y las hojas serían individuos. Para almacenar este árbol se van a utilizar dos clases: *Branch* y *Taxonomy*. La clase *Branch* representa una rama del árbol, por ejemplo, *{Selección, Portería}* o *{Portería, Casillas}*. La clase *Taxonomy* contiene la lista de ramas del árbol. Desarrolle el ETDS que genere la estructura de datos asociada a una taxonomía.

```
public class Taxonomy {
  private String name;
  private Branch[] list;

  public Taxonomy (String name) { ...  }
  public void addBranch(Branch b) { ... }
}

public class Branch {
   private String up;
   private String down;

   public Branch(String up, String down) { ... }
}
```

Ejercicio 5.18

La siguiente gramática describe la sintaxis de un lenguaje de definición de árboles en el formato de la herramienta JavaCC:

```
void Arbol() :
{}
{
   <TREE>  <ID>  <LBRACE>
   Nodo() ( <COMMA>  Nodo() )* <RBRACE>
}

void Nodo() :
{}
{
  NodoHoja()  |  NodoInterno()
}

void NodoHoja() :
{}
{
   <LBRACKET>   <ID>  <RBRACKET>
}

void NodoInterno():
{}
{
   <ID>  <LBRACE>  Nodo() ( <COMMA>  Nodo() )* <RBRACE>
}
```

Para almacenar un árbol se van a utilizar dos clases: *Branch* y *ListedTree*. La clase *Branch* representa una rama del árbol. La clase *ListedTree* contiene la lista de ramas del árbol. Modifique la descripción de JavaCC para que el símbolo *Arbol()* devuelva la representación del árbol basada en esta estructura de datos.

```
public class ListedTree {
  private String name;
  private Branch[] list;

  public ListedTree (String name) { ...  }
  public void addBranch(Branch b) { ... }
}

public class Branch {
   private String up;
   private String down;

   public Branch(String up, String down) { ... }
}
```

Ejercicio 5.19

A continuación se muestra una versión EBNF de la gramática simplificada del lenguaje PGN:

Game → (*Tag*)+ (*Movement*)+ **result**

Tag → **lbracket id string rbracket**

Movement → (**number**)? **move**

(a) Realice las transformaciones necesarias para expresar la gramática en notación BNF.

(b) Realice las transformaciones necesarias para que la gramática cumpla la condición LL(1).

(c) Desarrolle un ETDS basado en la gramática BNF - LL(1) obtenida, que genere el árbol de sintaxis abstracta que representa la partida de ajedrez reconocida. Enumere los atributos asignados a cada símbolo de la gramática.

Nota: Para representar una partida de ajedrez se dispone de la clase *Game*. Para almacenar las etiquetas (*tags*) de la partida, la clase *Game* dispone del método *addTag(String, String)*. El primero de los argumentos de este método se refiere al identificador de la etiqueta (que se obtiene del token *id*) y el segundo al valor de la etiqueta (que se obtiene del token *string*). Para representar los movimientos se dispone de la clase *MovementItem*. Esta clase describe un elemento de una lista de movimientos. Cada elemento tiene un descriptor del movimiento (que se obtiene del token *move* y se introduce en el constructor) y una referencia al siguiente elemento de la lista (que se inicializa a *null* en el constructor y se asigna mediante el método *setNext(MovementItem)*). Para almacenar la lista de movimentos en el objeto *Game* se utiliza el método *setMovementList(MovementItem)*). Se asume que los tokens (símbolos terminales) disponen de un atributo llamado *lexema*, de tipo *String*, que contiene el lexema del token.

A continuación se describen las clases *Game* y *MovementItem*.

```
public class Game {
  private String[] tag_id;
  private String[] tag_value;
  private MovementItem list;

  public Game () { ... }
  public void addTag(String id, String value) { ... }
  public void setMovementList(MovementItem m) { this.list = m; }
}

public class MovementItem {
   private String description;
   private MovementItem next;

  public MovementItem(String desc) { description = desc; next = null; }
   public setNext(MovementItem n) { this.next = n; }
}
```

Ejercicio 5.20

La siguiente gramática describe la sintaxis de las expresiones regulares:

Expr → *Option MoreOptions*

MoreOptions → **or** *Option MoreOptions*

MoreOptions → λ

Option → *Base MoreBases*

MoreBases → *Base MoreBases*

MoreBases → λ

Base → **symbol**

Base → **lparen** *Expr* **rparen** *Oper*

Oper → λ

Oper → **star**

Oper → **plus**

Oper → **hook**

Desarrolle a partir de esta gramática un ETDS que genere el árbol de sintaxis abstracta que describa la expresión regular. La estructura de datos a utilizar se describe en la próxima página.

```
// Clase abstracta que describe una expresión regular
public abstract class  Expression {
}

// Clase que describe un símbolo del lenguaje
public class Symbol extends Expression {
  private char symbol;

  public Symbol(char s) {  this.symbol = s; }
}

// Clase que describe un lista de opciones " a |  b  | c "
public class OptionList extends Expression {
  private Expression[] list;

  public OptionList(Expression exp) {
    this.list = new Expression[1];
    this.list[0] = exp;
  }
  public void addOption(Expression exp) { ... }
}

// Clase que describe una concatenación de expresiones " a  b  c "
public class ConcatList extends Expression {
  private Expression[] list;

  public ConcatList(Expression exp) {
    this.list = new Expression[1];
    this.list[0] = exp;
  }
  public void concat(Expression exp) { ... }
}

// Clase que describe una operación de clausura " ( a )* ",
// clausura positiva " ( a )+ ", o una opcionalidad " ( a )? "
 public class Operation extends Expression {
  public static final int STAR = 1;
  public static final int PLUS = 2;
  public static final int HOOK = 3;

  private int operator;
  private Expression operand;

  public Operation(int op, Expression exp) {
    this.operartor = op;
    this.operand = exp;
  }
}
```

Ejercicio 5.21

La siguiente figura muestra una gramática que describe el operador potencia en el formato de JavaCC.

```
void Potencia() :
{}
{
   <NUM>  (  <ELEV>  <NUM>  )*
}
```

La potencia es un operador binario que tiene la particularidad de considerarse asociativo a la derecha. Esto quiere decir que una expresión del tipo

```
2 ^ 3 ^ 4
```

debe entenderse como

```
2 ^ ( 3 ^ 4 )
```

al contrario de lo que ocurre con la mayoría de operadores binarios como la suma, el producto, etcétera.

Considere las siguientes clases que permiten definir las potencias de números:

```
// Clase abstracta que describe una expresión aritmética
public abstract class  Expression {
}

// Clase que describe un número constante
public class Number extends Expression {
  private double value;

  public Number(double val) {  this.value = val; }
}

// Clase que describe la potencia entre dos expresiones
public class Power extends Expression {
  public Expression base;
  public Expression pow;

  public Power(Expression a, Expression b) {
    this.base = a;
    this.pow = b;
  }
}
```

Desarrolle un ETDS que genere el árbol de sintaxis abstracta asociado a una expresión formada por el operador potencia.

Ejercicio 5.22

La siguiente gramática describe la sintaxis de la definición de tipos en un lenguaje funcional:

Type → **type** **id** **eq** *TypeExpr* **semicolon**

TypeExpr → *BaseExpr* *Relations*

Relations → **rel** *BaseExpr* *Relations*

Relations → λ

BaseExpr → **id**

BaseExpr → *Array*

BaseExpr → *Tuple*

Array → **lbra** *TypeExpr* **rbra**

Tuple → **lpar** *TypeExpr* *MoreExpr* **rpar**

MoreExpr → **comma** TypeExpr *MoreExpr*

MoreExpr → λ

A continuación se muestran algunos ejemplos de definición de tipos:

```
type Complex = ( Double , Double );  // Definición de los nº complejos

type DoubleList = [ Double ];  // Lista de números

type ElementAt = Int -> [ Double ] -> Int ; // Tipo de la función elementAt
```

Desarrolle a partir de esta gramática un ETDS que genere el árbol de sintaxis abstracta que describa la expresión regular. La estructura de datos a utilizar se describe en la próxima página.

```
// Clase que describe una definición de un tipo de datos
public class Type {
  private String id;
  private TypeExpression expr;

  public Type(String id, TypeExpression expr) { ... }
}

// Clase abstracta que describe una expresión de tipos
public abstract class  TypeExpression {
   private TypeExpression nextRelation = null;

   public void setNextRelation(TypeExpression rel) {
     this.nextRelation = rel;
   }
}

// Clase que describe una referencia a un tipo ya existente
public class Reference extends TypeExpression {
  private Type type;

  public Reference(String id) {  ... }
}

// Clase que describe un array de un tipo base
public class Array extends TypeExpression {
  private TypeExpression base;

  public Array(TypeExpression expr) {
    this.base = expr;
  }
}

// Clase que describe una tupla de tipos
public class Tuple extends TypeExpression {
  private TypeExpression[] list;

  public Tuple(TypeExpression elem) {
    this.list = new TypeExpression[1];
    this.list[0] = elem;
  }

  public void addField(TypeExpression elem) { ... }
}
```

Ejercicio 5.23

La siguiente figura muestra una gramática LL(1) que reconoce una lista de números separados por coma.

Lista → **num** *SigueLista*

SigueLista → **coma num** *SigueLista*

SigueLista → λ

Partiendo de esta gramática, desarrolle un ETDS que genere una lista doblemente enlazada que contenga los números reconocidos. Para representar la lista se utilizarán objetos de la clase *DoubleListElement* que permiten describir un elemento de la lista, donde el campo *prev* de cada objeto contiene el enlace al elemento anterior, el campo *next* contiene el enlace al elemento posterior y el campo *value* contiene el valor del número almacenado.

```
public class DoubleListElement {

  /** Valor almacenado en el objeto */
  private double value;

  /**  Enlace al elemento anterior de la lista */
  private DoubleListElement prev;

  /** Enlace al elemento posterior de la lista */
  private DoubleListElement next;

  /** Constructor */
  public DoubleListElement(double val) {
    this.value = val;
    this.prev = null;
    this.next = null;
  }

  /** Asignación del enlace al elemento anterior */
  public void setPrev(DoubleListElement p) {
    this.prev = p;
  }

  /** Asignación del enlace al elemento posterior */
  public void setNext(DoubleListElement n) {
    this.next = n;
  }
}
```

Ejercicio 5.24

La siguiente gramática describe en el formato de JavaCC las expresiones vectoriales formadas por las operaciones de suma de vectores y producto de un escalar por un vector.

```
void Expression() :
{}
{
   Factor() ( ( <PLUS> | <MINUS> ) Factor() )*
}

void Factor() :
{}
{
  (
     <NUM> <PROD> Factor()
   | Vector()
   | <LPAREN> Expression() <RPAREN>
  )
}
```

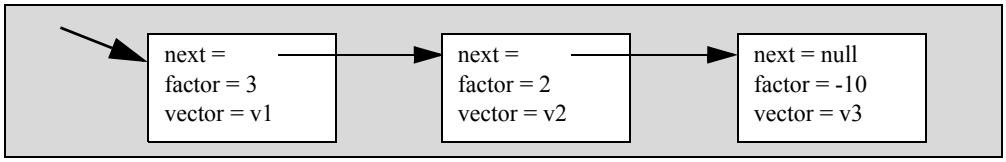
Para representar estas expresiones se pretenden utilizar listas de términos. Cada uno de estos términos representa el resultado del producto de un escalar por un vector y se describe por medio de un objeto de la clase *Term*. Considere que el símbolo Vector() que aparece en la gramática devuelve un objeto *Vect* que describe un vector.

```
public class Term {

   private double factor;
   private Vect vector;
   private Term next;

   public Term(double f, Vect v) {
     this.factor = f;
     this.vector = v;
     this.next = null;
   }

   public double getFactor() { return this.factor }
   public void setFactor(double f) { this.factor = f; }

   public Vect getVector() { return this.vcetor; }
   public void setVector(Vect v) { this.vector = v; }

   public Term getNext() { return this.next; }
   public void setNext(Term n) { this.next = n; }
}
```

Modifique la descripción de JavaCC para que el símbolo *Expression()* devuelva una lista de términos formada por objetos de la clase Term. Por ejemplo, para la entrada

```
3 * v1 + 2 * ( v2 - 5 * v3 )
```

la lista de términos a generar es la siguiente:



Ejercicio 5.25

La siguiente gramática describe expresiones formadas con las operaciones producto y potencia de números:

Expresión → *Factor Producto*

Producto → **prod** *Factor Producto*

Producto → λ

Factor → *Base Potencia*

Potencia → **elev** *Base Potencia*

Producto → λ

Base → **num**

Base → **lparen** *Expresión* **rparen**

Considere las siguientes clases que permiten definir expresiones con estos operadores:

```
// Clase abstracta que describe una expresión aritmética
public abstract class  Expression {
}

// Clase que describe un número constante
public class Number extends Expression {
  private double value;

  public Number(double val) {  this.value = val; }
}

// Clase que describe la potencia entre dos expresiones
public class Power extends Expression {
  public Expression base;
  public Expression pow;

  public Power(Expression a, Expression b) {  this.base = a;   this.pow = b;  }
}

// Clase que describe el producto entre dos expresiones
public class Product extends Expression {
  public Expression left;
  public Expression right;

  public Product(Expression a, Expression b) {  this.left = a; this.right = b;  }
}
```

Desarrolle un ETDS que genere el árbol de sintaxis abstracta asociado a una expresión formada por los operadores producto y potencia. Es importante tener en cuenta que la potencia es un operador asociativo a la derecha, mientras que el producto es un operador asociativo a la izquierda.

Ejercicio 5.26

La siguiente gramática describe expresiones formadas mediante la combinación de strings por medio de dos operadores: la concatenación y la disyunción. El operador de concatenación se define como un operador asociativo a la derecha, mientras que el operador de disyunción se define como un operador asociativo a la izquierda.

Expresión → *Cadena* *Concatenaciones*

Concatenaciones → **concat** *Cadena* *Concatenaciones*

Concatenaciones → λ

Cadena → *Base* *Disyunciones*

Disyunciones → **disjunc** *Base* *Disyunciones*

Disyunciones → λ

Base → **string**

Base → **lparen** *Expresión* **rparen**

Considere las siguientes clases que permiten definir expresiones con estos operadores:

```
// Clase abstracta que describe una expresión
public abstract class  Expresion {
}

// Clase que describe una cadena
public class Cadena extends Expresion {
  private String value;

  public Number(String val) {  this.value = val; }
}

// Clase que describe la concatenación entre dos expresiones
public class Concat extends Expresion {
  public Expression left;
  public Expression right;

  public Concat(Expression a, Expression b) {  this.left = a;   this.right = b;  }
}

// Clase que describe la disyunción entre dos expresiones
public class Disjunc extends Expresion {
  public Expression left;
  public Expression right;

  public Disjunc(Expression a, Expression b) {  this.left = a; this.right = b;  }
}
```

Desarrolle un ETDS que genere el árbol de sintaxis abstracta asociado a una expresión formada por estos operadores.

Ejercicio 5.27

La siguiente gramática permite describir expresiones vectoriales formadas por dos operaciones: la suma de vectores y el producto de un escalar por un vector.

Expr → *Factor SigueExpr*

SigueExpr → **plus** *Factor SigueExpr*

SigueExpr → **minus** *Factor SigueExpr*

SigueExpr → λ

Factor → *Vector*

Factor → **num prod** *Factor*

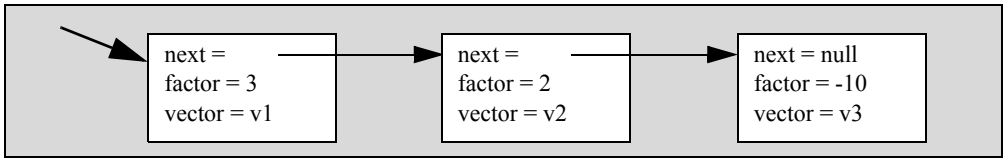
Factor → **lpar** *Expr* **rpar**

Para representar estas expresiones se pretenden utilizar listas de términos. Cada uno de estos términos representa el resultado del producto de un escalar por un vector y se describe por medio de un objeto de la clase *Term* (que se describe en la página siguiente). La lista representa la suma de todos esos términos. Considere que el símbolo *Vector* que aparece en la gramática devuelve un objeto *Vect* que describe un vector.

Modifique la gramática anterior para que el símbolo *Expr* devuelva una lista de términos formada por objetos de la clase *Term*. Por ejemplo, para la entrada

```
3 * v1 + 2 * ( v2 - 5 * v3 )
```

la lista de términos a generar es la siguiente:



Ejercicio 5.28

La siguiente gramática permite describir los pedidos de una cafetería.

Pedido → *Lista*

Lista → *Término SigueLista*

SigueLista → **plus** *Término SigueLista*

SigueLista → λ

Término → **producto**

Término → **num prod** *Término*

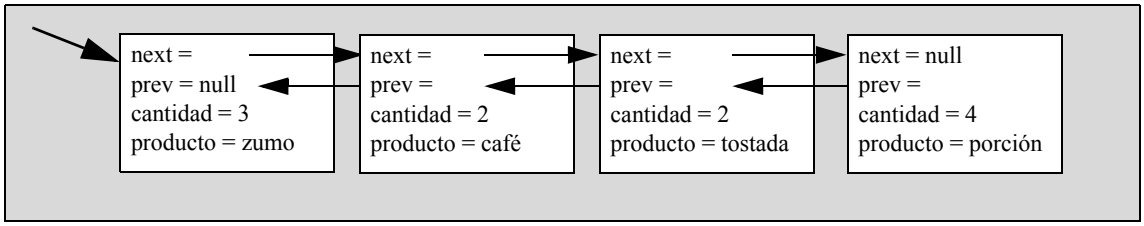
Término → **lpar** *Lista* **rpar**

Para representar estos pedidos se pretenden utilizar listas de términos doblemente enlazadas. Cada uno de estos términos representa un producto del pedido y la información incluye el número de unidades de ese producto (por ejemplo, tres cafés) y se describe por medio de un objeto de la clase *Term* (que se describe en la página siguiente).

Modifique la gramática anterior para que el símbolo *Pedido* devuelva una lista de términos formada por objetos de la clase *Term*. Por ejemplo, para la entrada

3 * zumo + 2 * (café + tostada + 2 * porción)

la lista de términos a generar es la siguiente:



Ejercicio 5.29

La gramática presentada a continuación permite describir en el formato de JavaCC una representación gráfica bidimensional por medio de particiones horizontales y verticales.

```
void Figura() :
{}
{
  <FIGURE>
  <LBRACKET> <ENTERO> <COMA> <ENTERO> <RBRACKET>
  Particion()
}

void Particion() :
{}
{
  Horizontal()  |  Vertical() | Color()
}

void Horizontal() :
{}
{
  <HORIZONTAL> <LBRACKET> <REAL> <RBRACKET>
  <LBRACE> Particion() <COMA> Particion() <RBRACE>
}

void Vertical() :
{}
{
  <VERTICAL> <LBRACKET> <REAL> <RBRACKET>
  <LBRACE> Particion() <COMA> Particion() <RBRACE>
}

void Color() :
{}
{
    <COLOR>
    <LPAREN> <ENTERO> <COMMA> <ENTERO>
                      <COMMA> <ENTERO> <RPAREN>
}
```

La dimensión contiene dos números enteros que expresan en pixels la anchura y altura de la figura descrita. El factor contiene un número real entre 0 y 1 que expresa el tamaño de la división como tanto por uno (p.e., el factor 0.3 representa una división del espacio del 30%). El color contiene la representación RGB de relleno del espacio resultante. Por ejemplo, la siguiente descripción representa un cuadrado rojo de 20x20 pixels:

```
figura [20,20] color (256,0,0)
```

Otro ejemplo: un rectángulo de 30x10 pixels dividido horizontalmente en dos trozos. El primer trozo mide 12x10 pixels y es de color azul. El segundo mide 18x10 pixels y es verde.

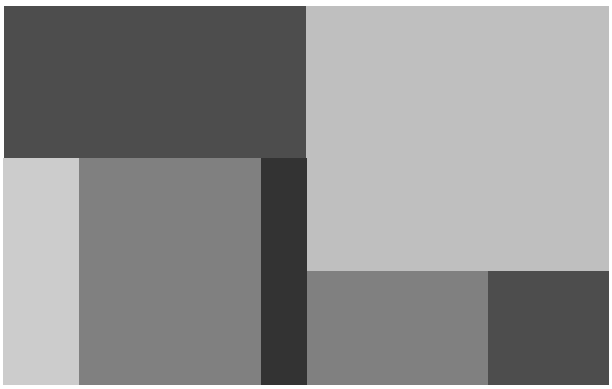
```
figura [30,10] horizontal [0.4] {
   color (0,0,256),
   color (0,256,0)
}
```

Construya un ETDS que permita generar una estructura de datos a partir de la gramática planteada. Dicha estructura de datos se construirá por medio de las siguientes clases:

```
public class Figura {
  int width;
  int height;
  Particion desc;

  Figura(int w, int h, Particion p) {
    this.width = w;
    this.height = h;
    this.desc = p;
  }
}

public abstract class Particion {
}

public class Horizontal extends Particion {
  Particion left, right;

  public Horizontal(Particion l, Particion r) {
     this.left = l;
     this.right = r;
  }
}

public class Vertical extends Particion {
  Particion top, bottom;

  public Vertical(Particion t, Particion b) {
     this.top = t;
     this.bottom = b;
  }
}

public class Rectangulo extends Particion {
  int width, height;
  int red, green, blue;

  public Rectangulo(int w, int h, int r, int g, int b) {
    this.width = w; this.height = h;
    this.red = r; this.green = g; this.blue = b;
  }
}
```

La siguiente figura muestra un ejemplo más complejo de descripción de una representación gráfica en el formato planteado.

```
figura [80,50] horizontal  [0.5] {
    vertical [0.4] {
      color (180,180,180) ,
      horizontal [0.25] {
          color (51,51,51),
          horizontal [0.8] {
              color (128,128,128),
              color (205,205,205)
          }
      }
    },
    vertical [0.7] {
       color (64,64,64) ,
       horizontal [0.6] {
         color (128,128,128),
         color (180,180,180)
       }
    }
}
```



Ejercicio 5.30

La siguiente gramática permite describir los pedidos de una cafetería.

Pedido → *Lista*

Lista → *Término* (**plus** *Término*)*

Término → **producto**

Término → **num** **prod** *Término*

Término → **lpar** *Lista* **rpar**

Para representar estos pedidos se pretenden utilizar listas de términos doblemente enlazadas. Cada uno de estos términos representa un producto del pedido y la información incluye el número de unidades de ese producto (por ejemplo, tres cafés) y se describe por medio de un objeto de la clase *Term*.

```
public class Term {

   private double cantidad;
   private String producto;
   private Term next;
   private Term prev;

   public Term(double c, String p) {
     this.cantidad = c;
     this.producto = p;
     this.next = null;
     this.prev = null;
   }

   public double getCantidad() { return this.cantidad; }
   public void setCantidad(double c) { this.cantidad = c; }

   public String getProducto() { return this.producto; }
   public void setProducto(String p) { this.producto = p; }

   public Term getNext() { return this.next; }
   public void setNext(Term t) { this.next = t; }

   public Term getPrev() { return this.prev; }
   public void setPrev(Term t) { this.prev = t; }
}
```

Modifique la gramática anterior para que el símbolo *Pedido* devuelva una lista de términos formada por objetos de la clase *Term*. Por ejemplo, para la entrada

3 * zumo + 2 * (café + tostada + 2 * porción)

la lista de términos a generar es la siguiente: