Árbol Binario (Binary Tree o BT)

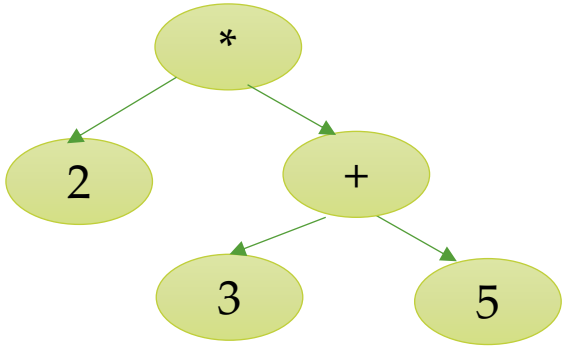
Estructura de datos formada por nodos, donde cada nodo o está vacío o tiene 3 componentes: datos, subárbol izquierdo y subárbol derecho. Existe un nodo distinguido llamado raíz.

Aplicaciones

Ejemplo 1: una vez más los compiladores

Las expresiones formadas por operadores unarios/binarios pueden representarse con BT, donde las expresiones más anidadas se deberán evaluar primero.

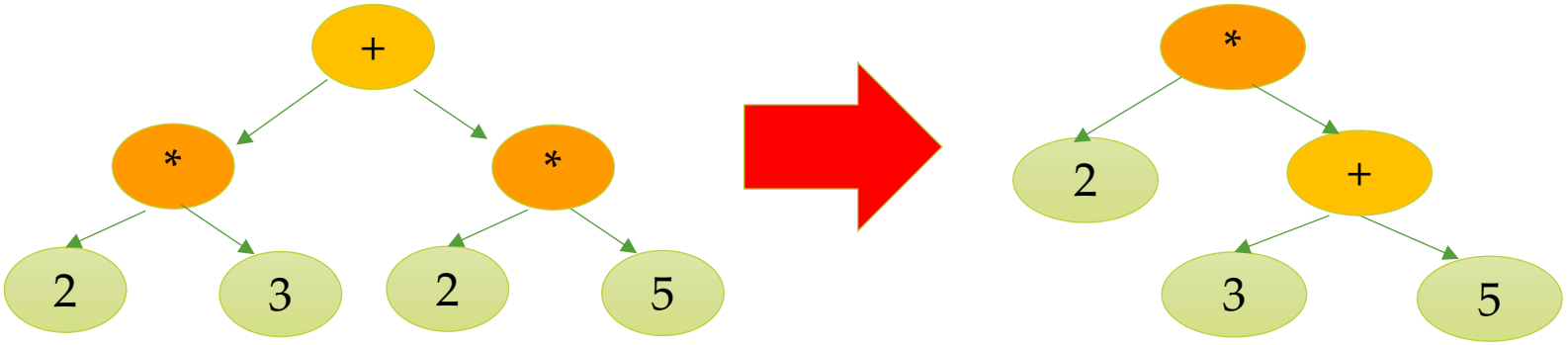
2 * (3 + 5)



Aplicaciones

Ejemplo 2: una vez más los compiladores

Pueden usar estrategias para optimizar las expresiones a la hora de evaluarlas (re estructurar el BT).



Aplicaciones

Ejemplo 3:

Cualquier cosa que tenga una representación jerárquica. Ej: jefe_de en una organización

Aplicaciones

Ejemplo 4:

Si el BT estuviera ordenado, podría usarse como soporte para índices (a este tipo de BT lo veremos más adelante)

Árbol Binario de Expresiones

Se utiliza para representar expresiones algebraicas. Los nodos internos representan operadores binarios o unarios. Las hojas representan los operandos, es decir, constantes y variables.

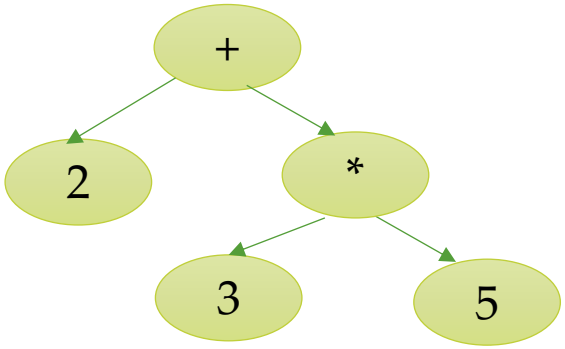
Según cómo se recorra el árbol (traversal) in-order, pre-order o post-order, se obtiene una expresión infija, prefija o postfija, respectivamente.

Características

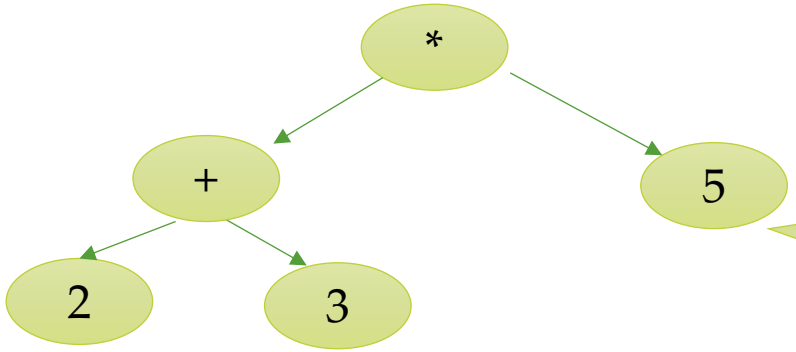
- Permite representar expresiones en notación infija (no es un árbol ordenado por el contenido)

- Así como usábamos una **pila y una tabla de precedencia de operadores** para pasar de una expresión en notación infija a postfija (para eliminar ambigüedad) y luego **con una pila** evaluábamos la expresión, ahora también a partir de una expresión, por ejemplo infija, **construiremos el árbol de expresiones** asociado y lo evaluaremos para devolver el valor de la expresión.

Ej: Este árbol representa la expresión infija (2 + (3 * 5))



Ej: este árbol representa la expresión infija ((2 + 3) * 5)



¡No se representan los paréntesis!

Aclaración

Para evitar la discusión sobre la precedencia de operadores, vamos a aceptar solo expresiones infijas que tengan paréntesis. Los operadores son todos binarios: +, -, *, /, ^ y es obligatorio usar paréntesis para toda expresión.

Para el input vamos a pedir que finalice en \n. Los espacios serán los separadores de tokens.

Formalmente

Una expresión aritmética E está dada por las siguientes reglas de derivación:

E -> (E + E)

E -> (E – E)

E -> (E * E)

E -> (E / E)

E -> (E ^ E)

E -> cte

Casos de Uso A

- new ExpTree("(2 + 3) \n");

E -> (E + E)
E -> (E – E)
E -> (E * E)
E -> (E / E)
E -> (E ^ E)
E -> cte

Casos de Uso A

- new ExpTree("(2 + 3) \n");

E -> (E op E)
E -> cte

Con op: + - * / ^

Casos de Uso A

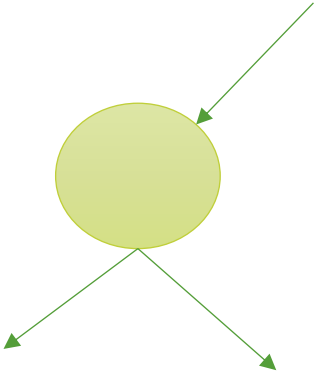
- new ExpTree("(2 + 3) \n");

E -> (E op E)
E -> cte

Con op: + - * / ^

Aplico

E -> (E op E)



Casos de Uso A

- new ExpTree("(2 + 3) \n");

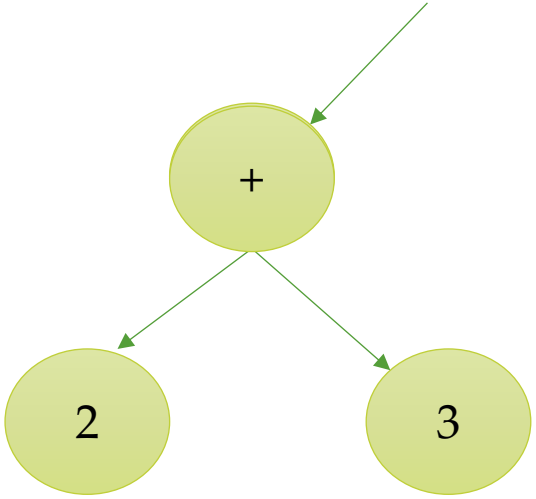
E -> (E op E)
E -> cte

Con op: + - * / ^

Aplico

E -> (E op E)

Y cada E -> cte



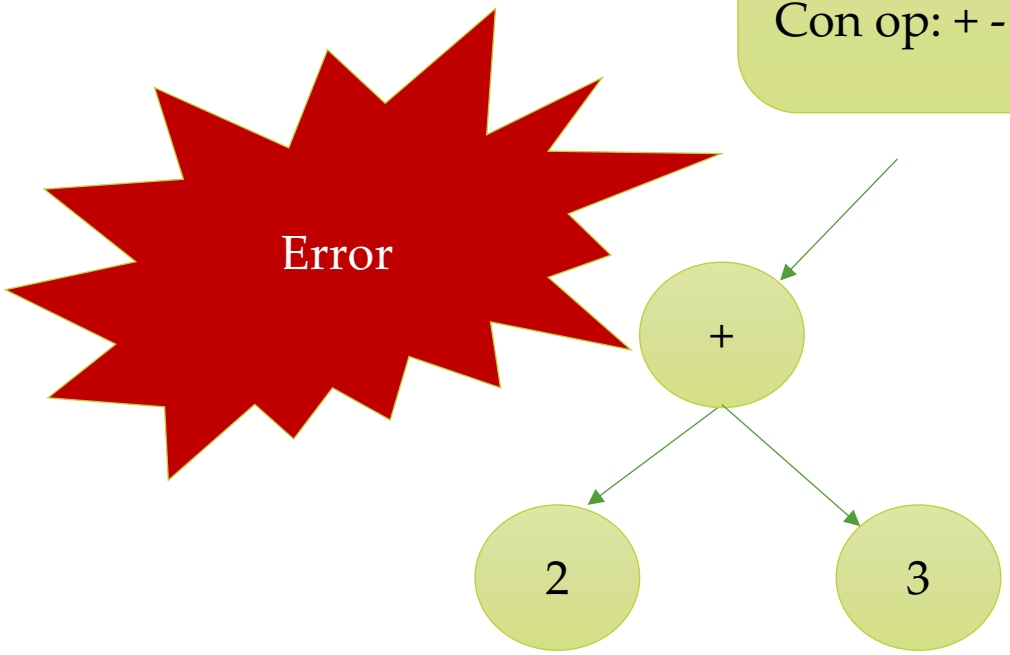
Casos de Uso B

- new ExpTree("(2 + 3))\n");

E -> (E op E)
E -> cte

Con op: + - * / ^

Aplico
E -> (E op E)
Y cada E -> cte



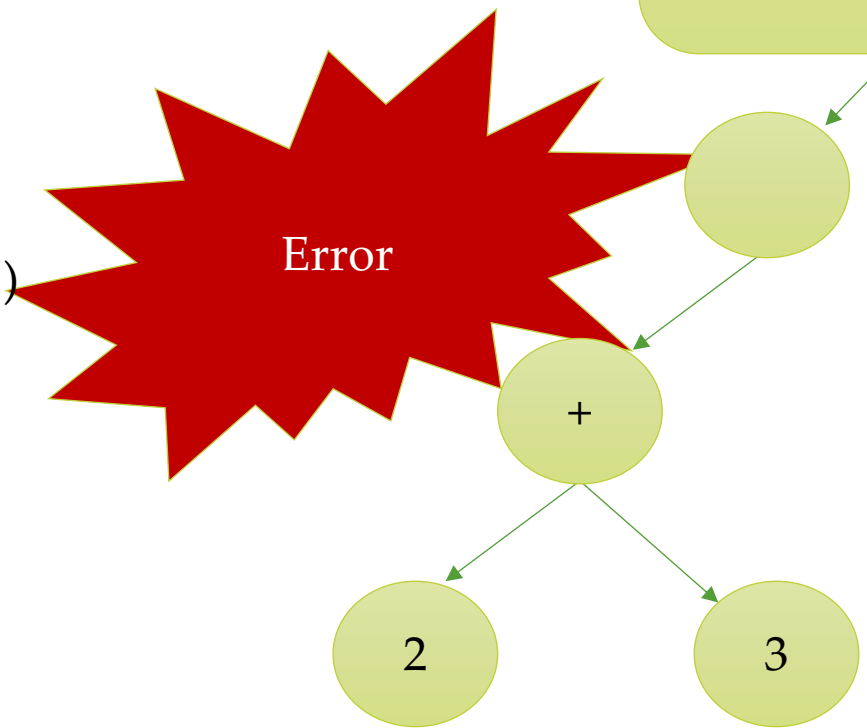
Casos de Uso C

- new ExpTree("((2 + 3)) \n");

E -> (E op E)
E -> cte

Con op: + - * / ^

Aplico
E -> (E op E)
primer E -> (E op E)



Casos de Uso D

- new ExpTree(“((2 + 3.5) * (-5 / -1))\n");

E -> (E op E)
E -> cte

Con op: + - * / ^

Casos de Uso D

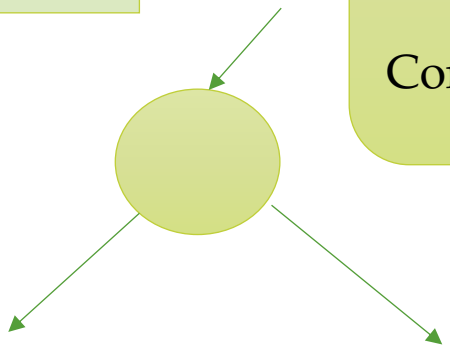
- new ExpTree(“((2 + 3.5) * (-5 / -1))\n");

E -> (E op E)
E -> cte

Con op: + - * / ^

Aplico

E -> (E op E)



Casos de Uso D

- new ExpTree(“((2 + 3.5) * (-5 / -1))\n");

E -> (E op E)
E -> cte

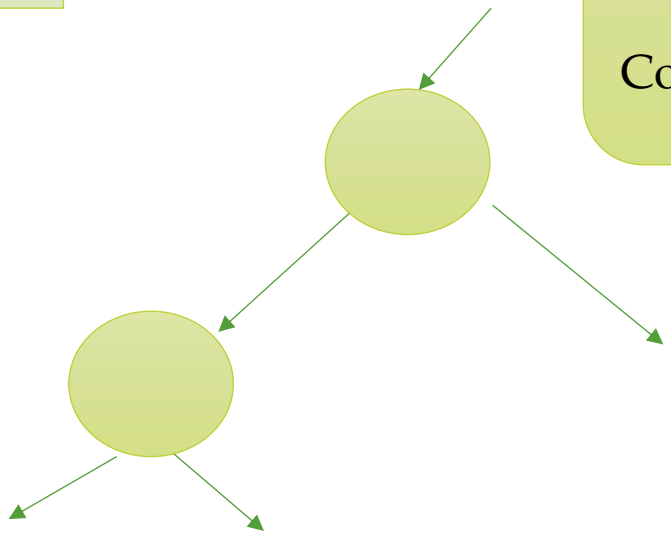
Con op: + - * / ^

Aplico

E -> (E op E)

El primer E se expande

E-> (E op E)



Casos de Uso D

- new ExpTree("((2 + 3.5) * (-5 / -1))\n");

E -> (E op E)
E -> cte

Con op: + - * / ^

Aplico

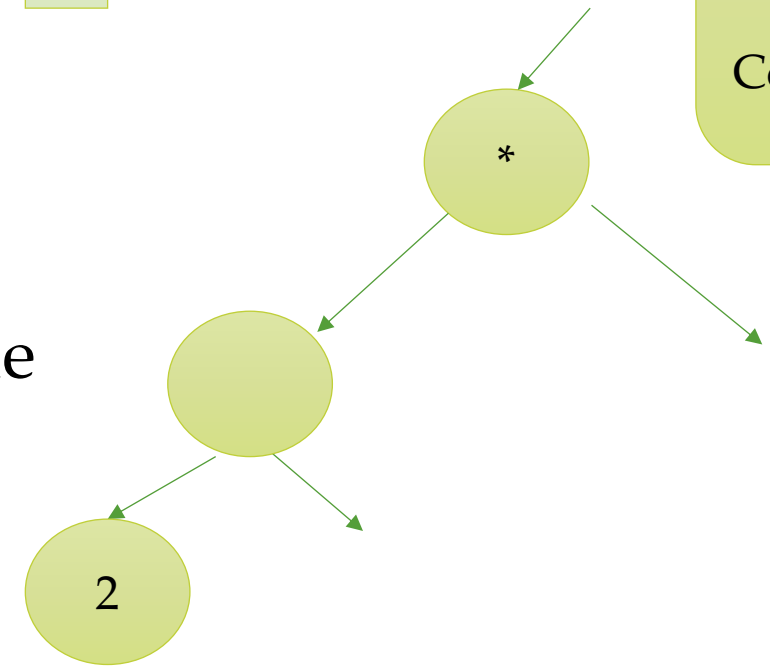
E -> (E op E)

El primer E se expande

E-> (E op E)

Ambos 2 aplican

E -> Cte



Casos de Uso D

- new ExpTree("((2 + 3.5) * (-5 / -1))\n");

E -> (E op E)
E -> cte

Con op: + - * / ^

Aplico

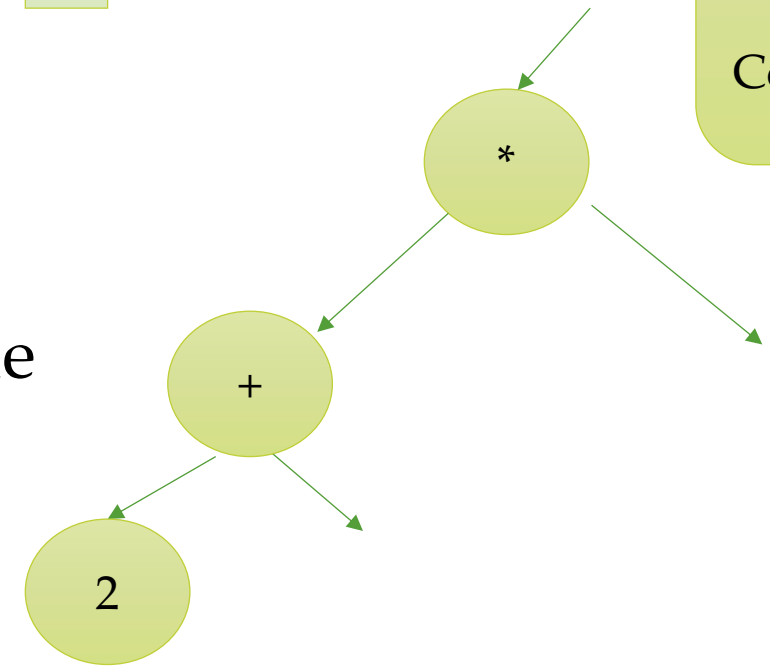
E -> (E op E)

El primer E se expande

E-> (E op E)

Ambos 2 aplican

E -> Cte



Casos de Uso D

- new ExpTree("((2 + 3.5) * (-5 / -1))\n");

E -> (E op E)
E -> cte

Con op: + - * / ^

Aplico

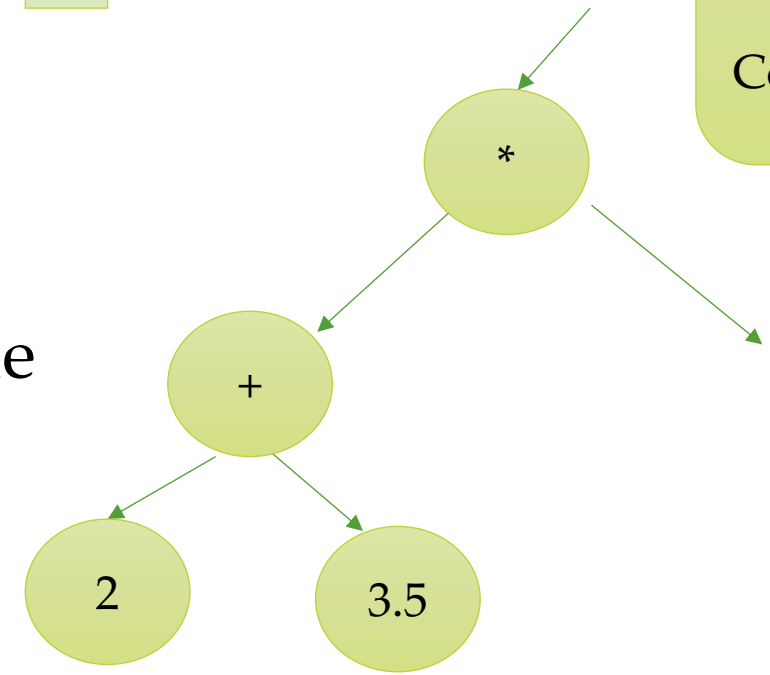
E -> (E op E)

El primer E se expande

E-> (E op E)

Ambos 2 aplican

E -> Cte



Casos de Uso D

- new ExpTree("((2 + 3.5) * (-5 / -1))\n");

E -> (E op E)
E -> cte

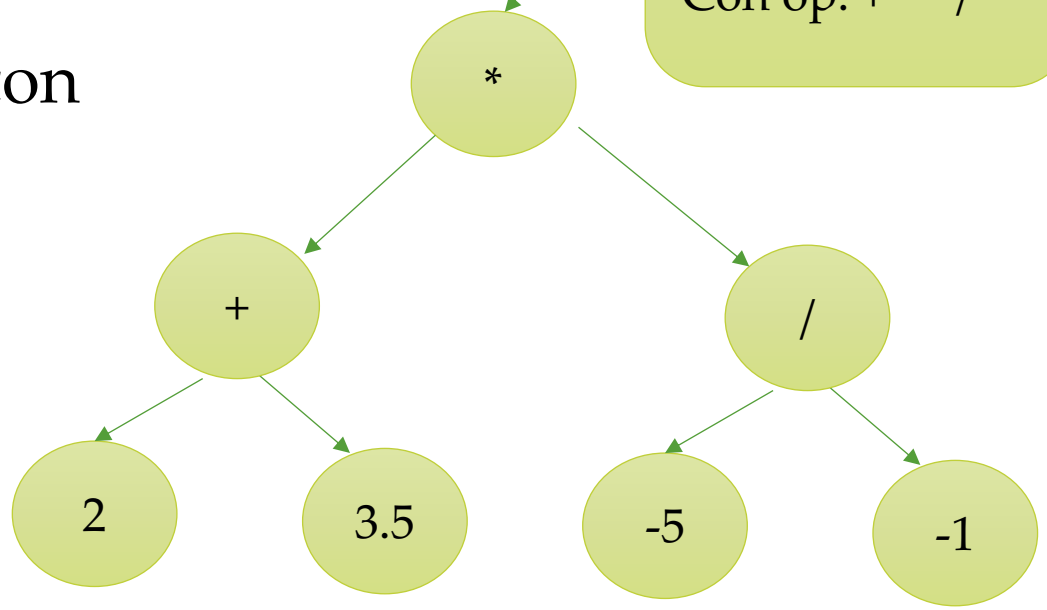
Con op: + - * / ^

Idem en la otra parte con

E -> (E op E)

Y ambos 2 por

E -> cte

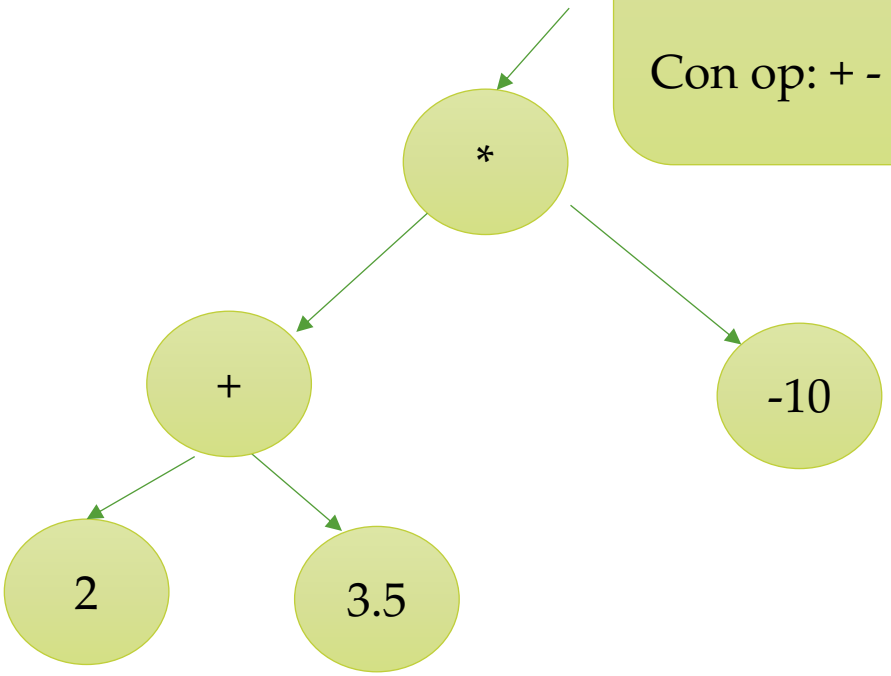


Casos de Uso E

- new ExpTree("((2 + 3.5) * -10)\n");

E -> (E op E)
E -> cte

Con op: + - * / ^



Ejercicio

Queremos que se construya el árbol de expresiones.

myExp= new ExpTree("((2 + 3.5) * -10)\n");

O que lance exception si no es correcta la expresión infija.

TP 5 – Ejer 2

Generar una aplicación que genere el árbol de expresiones correctas e incorrectas

(bajar de campus los templates)

Expresiones correctas:

```
myExp= new ExpTree("( -2.5 + 3 )    \n");
myExp= new ExpTree("(  (  2 + 3.5  ) * -10 )\n");
new ExpTree("(  (  2 + 3.5 ) * ( -5 / -1)  )\n");
```

Expresiones incorrectas:

```
new ExpTree("(  2 + 3  )  )\n");
new ExpTree("(  ( 2 +  3 )  ) \n");
new ExpTree(" ( 2 & 3 )   \n");
```

```java
static final class Node {
    private String data;
    private Node left, right;

    private Scanner lineScanner;

    public Node(Scanner theLineScanner) {
        lineScanner= theLineScanner;

        Node auxi = buildExpression();
        data= auxi.data;
        left= auxi.left;
        right= auxi.right;

        if (lineScanner.hasNext() )
            throw new RuntimeException("Bad expression");
    }

    private Node()  {
    }

    private Node buildExpression() {
       COMPLETE!!!!!
    }
} // class Node

} // class ExpTree
```

E -> (E op E)
E -> cte

Con op: + - * / ^

Tip. Es recursivo.
Además de
hasNext()
Y
next()
Usar:
hasNext("\\(")

Seguimiento del método recursivo buildExpression() para "((3 * (5 - 10.2)) - 2)"

E -> (E op E)
E -> cte

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```

E -> (E op E)
E -> cte

Con op: + - * / ^

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```

E -> (E op E)
E -> cte

Con op: + - * / ^

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
            lineScanner.next(); // lo consumo

            n.left = buildExpression(); // subexpression

            // operator
            if (!lineScanner.hasNext())
                throw new RuntimeException("missing or invalid operator");
            n.data = lineScanner.next();
            if (!Utils.isOperator(n.data))
                throw new RuntimeException("missing or invalid operator");

            // subexpression
            n.right = buildExpression();

            // ) expected
            if (lineScanner.hasNext("\\)")) {
                // lo consumo
                lineScanner.next();
            } else {
                throw new RuntimeException("missing )" );
            }

            return n;
        }

        // constant
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
            throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```

E -> (E op E)
E -> cte

Con op: + - * / ^

((3 * (5 - 10.2)) - 2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```

((3 * (5 - 10.2)) - 2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```

(3 * (5 - 10.2)) - 2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```

(3 * (5 - 10.2)) - 2)

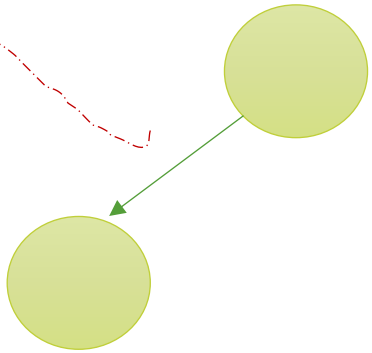
```
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```

pendiente 1

(3 * (5 - 10.2)) - 2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```

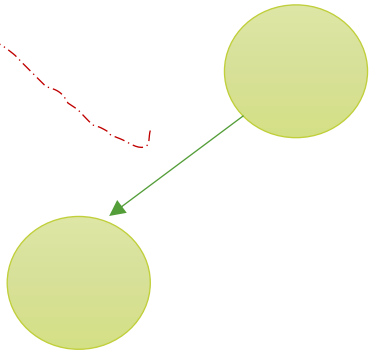
pendiente 1



(3 * (5 - 10.2)) - 2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```

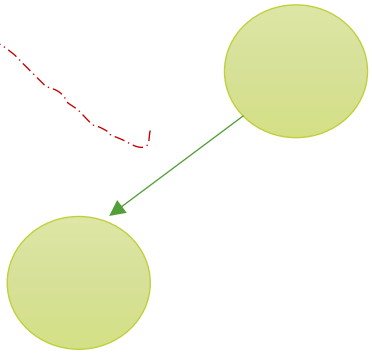
pendiente 1



3 * (5 - 10.2)) - 2)

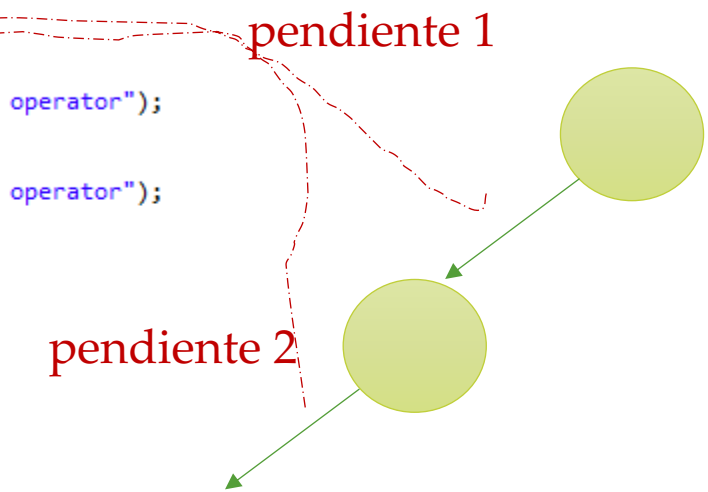
```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```

pendiente 1



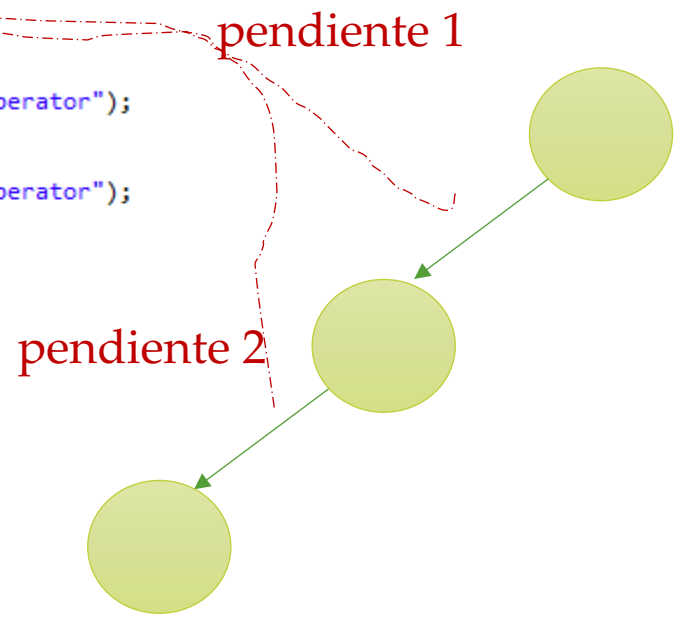
3 * (5 - 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



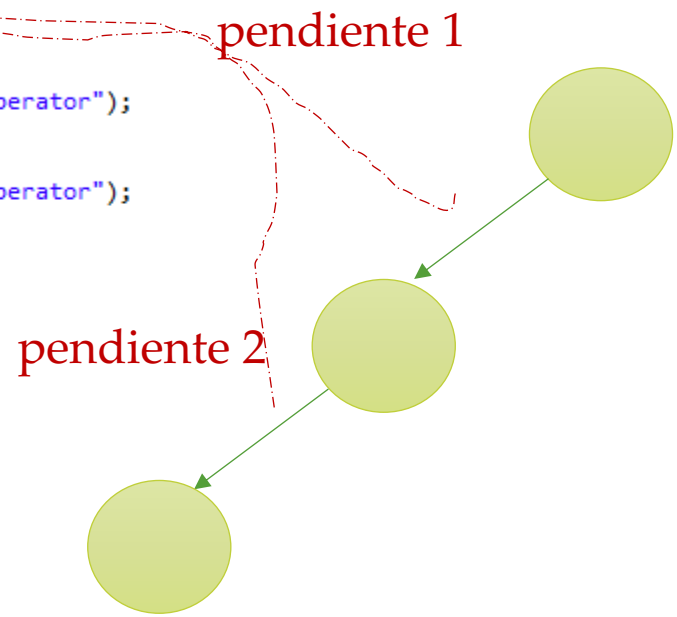
3 * (5 - 10.2)) - 2)

```
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```



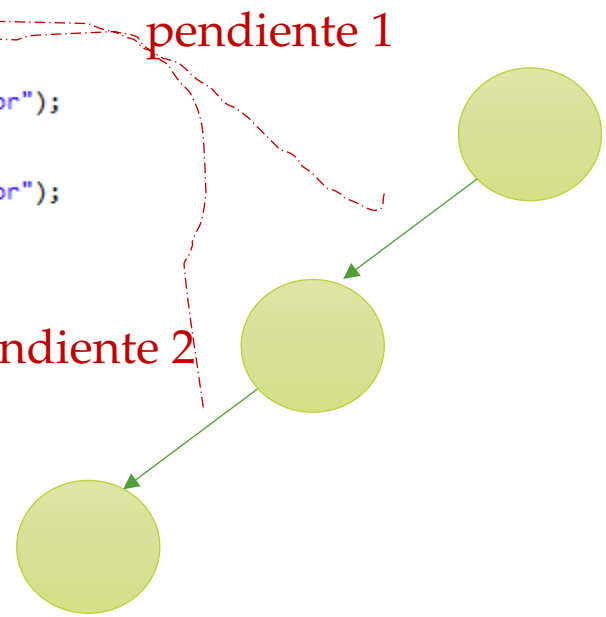
3 * (5 - 10.2)) - 2)

```
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```



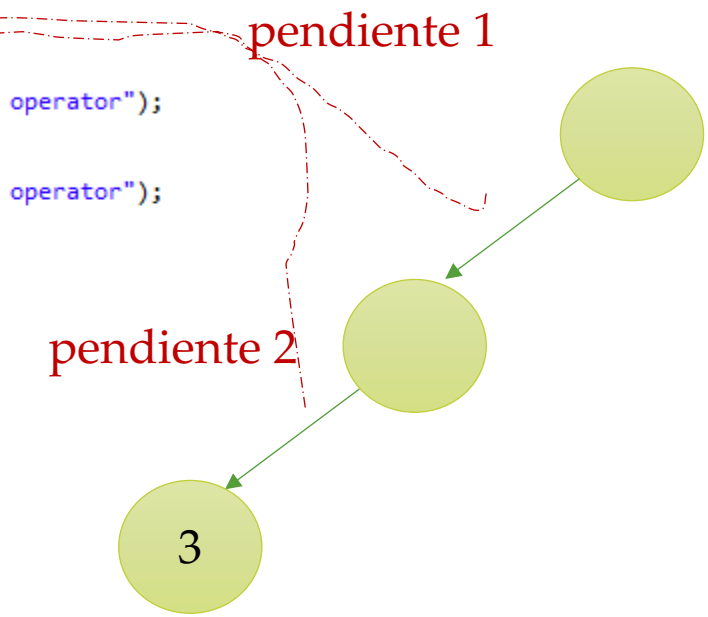
3 * (5 - 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



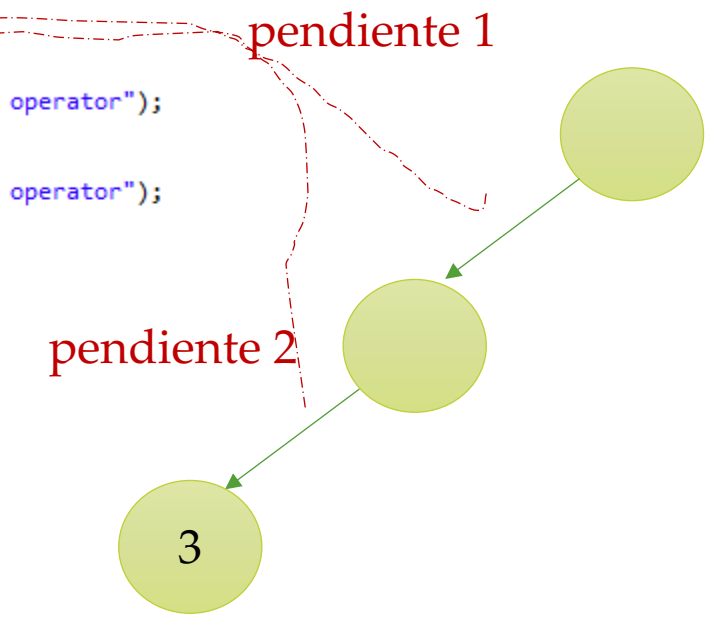
* (5 - 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
     }
```



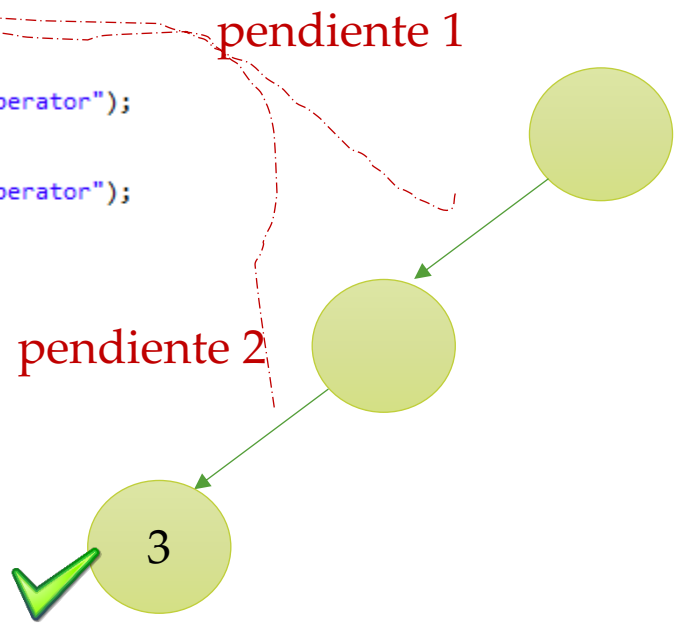
* (5 - 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



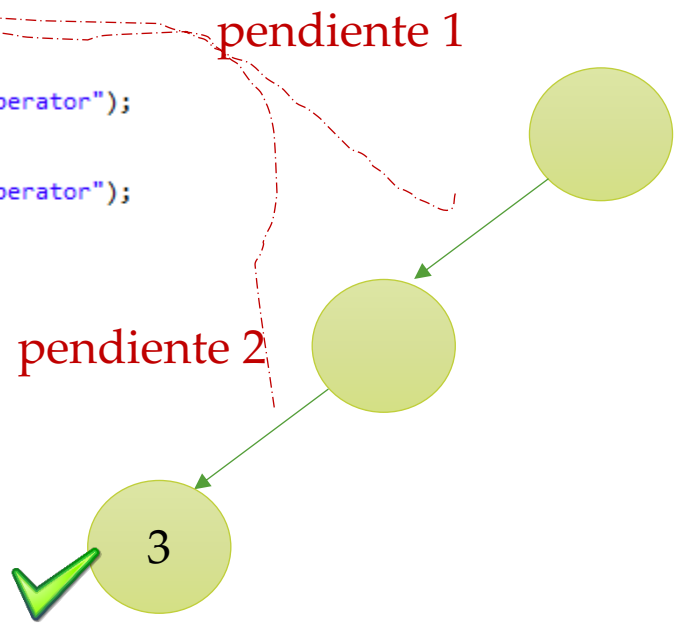
* (5 - 10.2)) - 2)

```
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



* (5 - 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



* (5 - 10.2)) - 2)

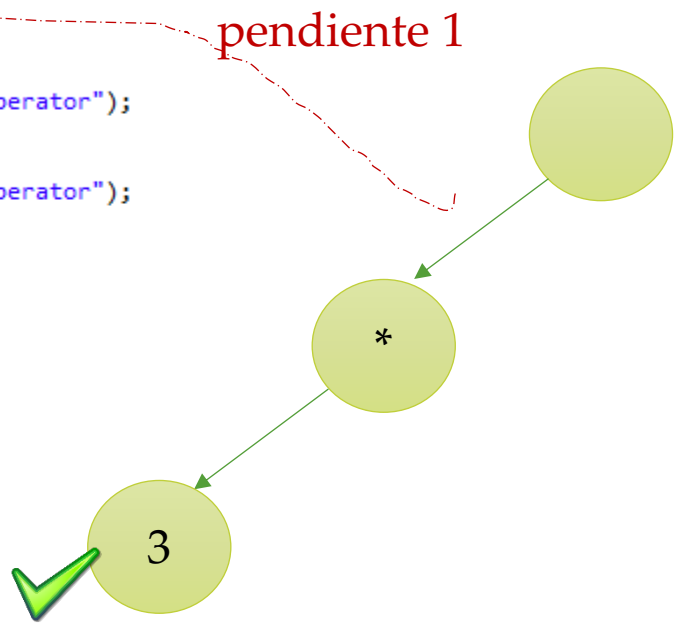
```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```

pendiente 1

3

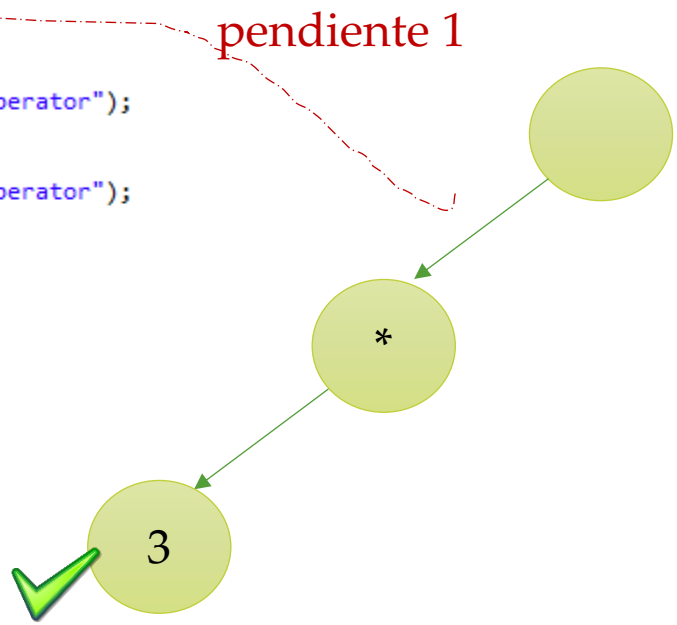
(5 - 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



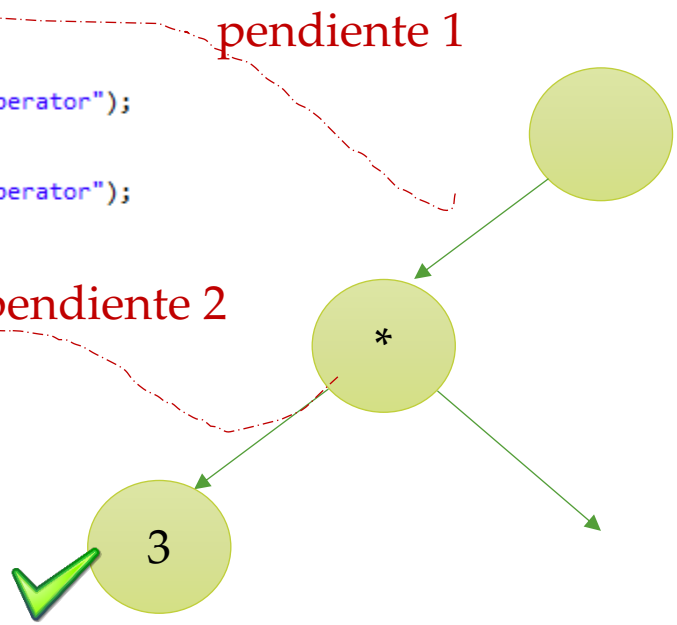
(5 - 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



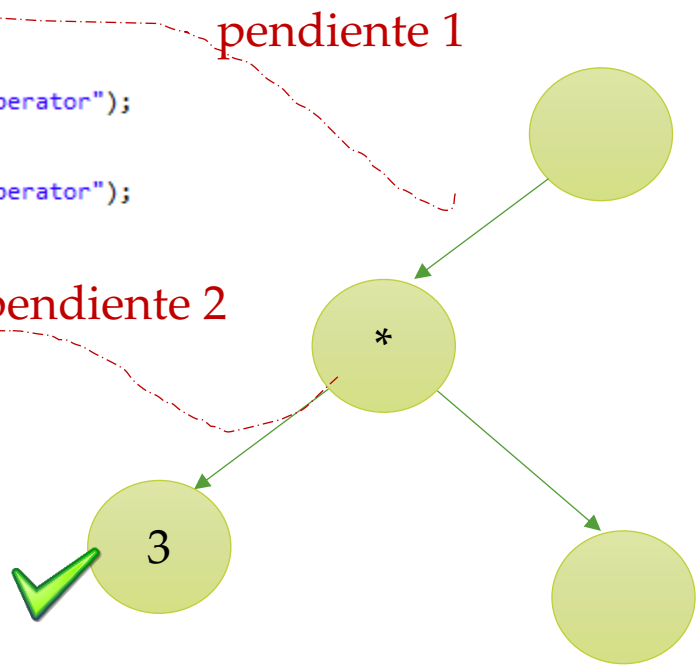
(5 - 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



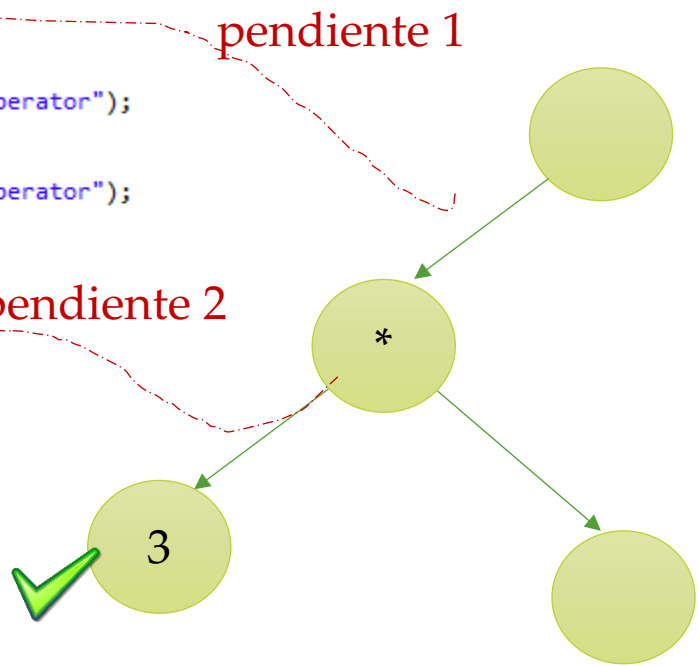
(5 - 10.2)) - 2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```



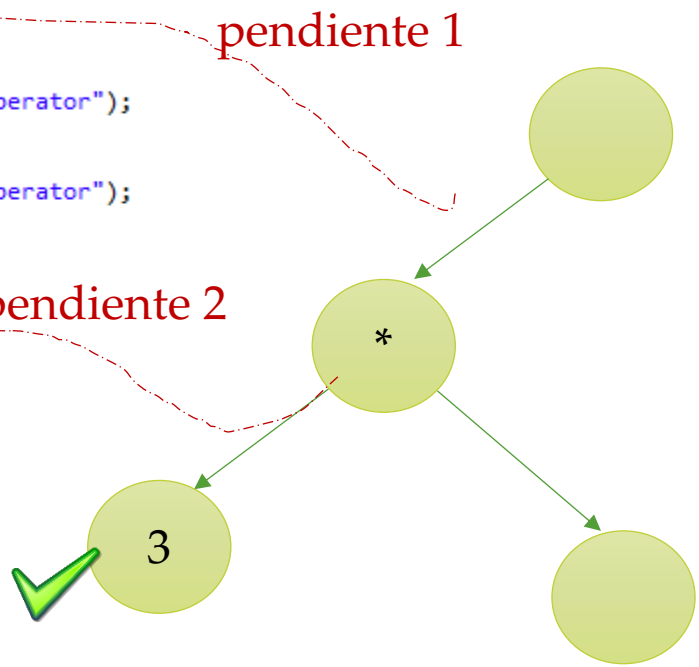
(5 - 10.2)) - 2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```



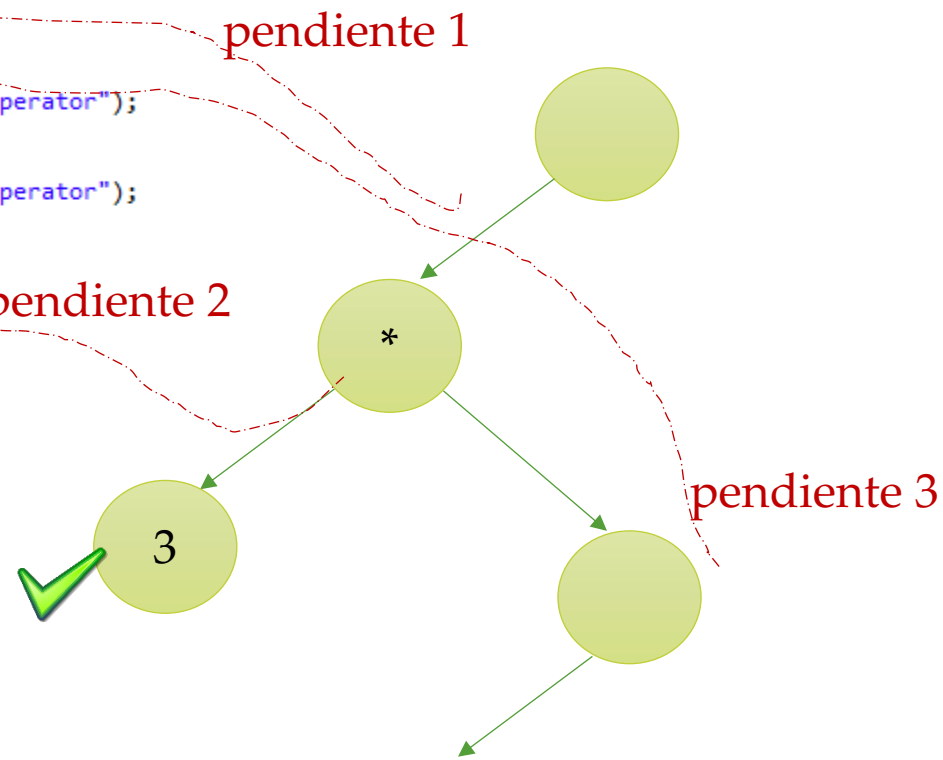
5 - 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



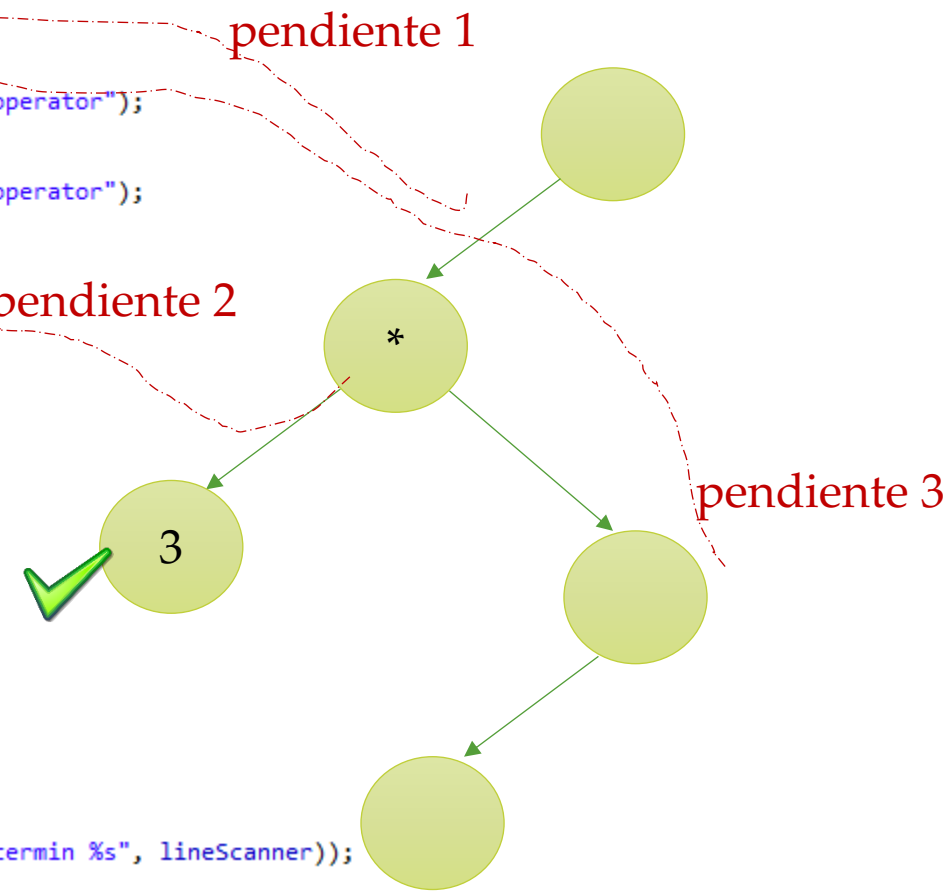
5 - 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



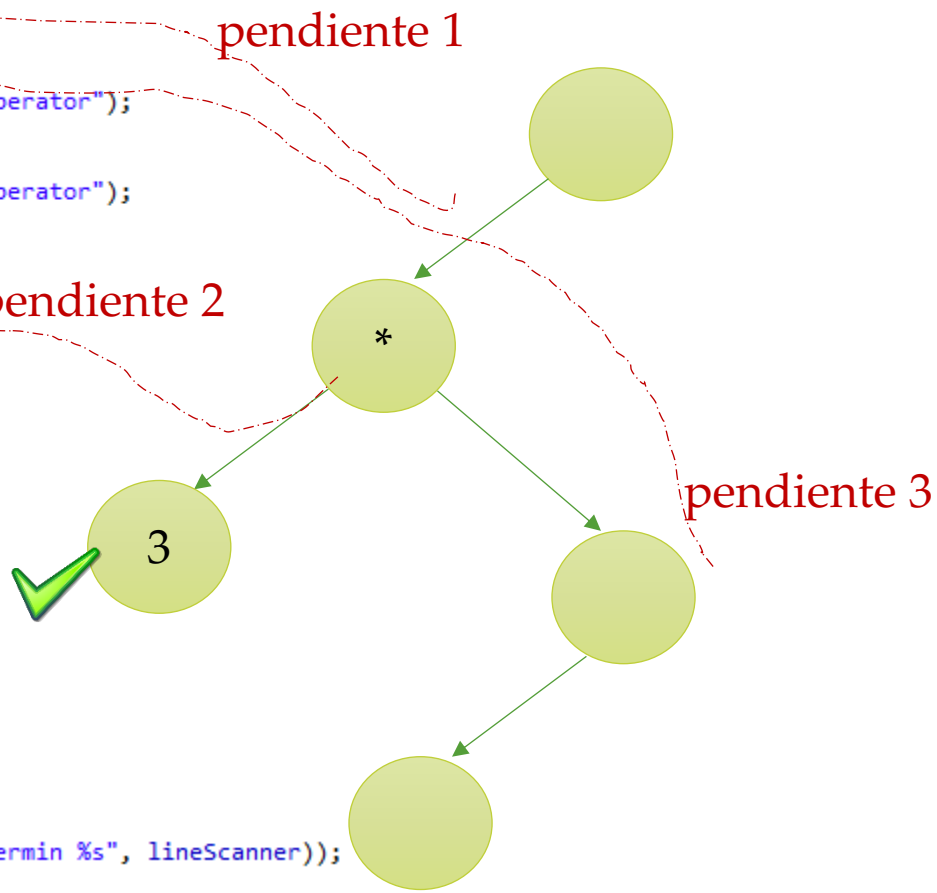
5 - 10.2)) - 2)

```
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```



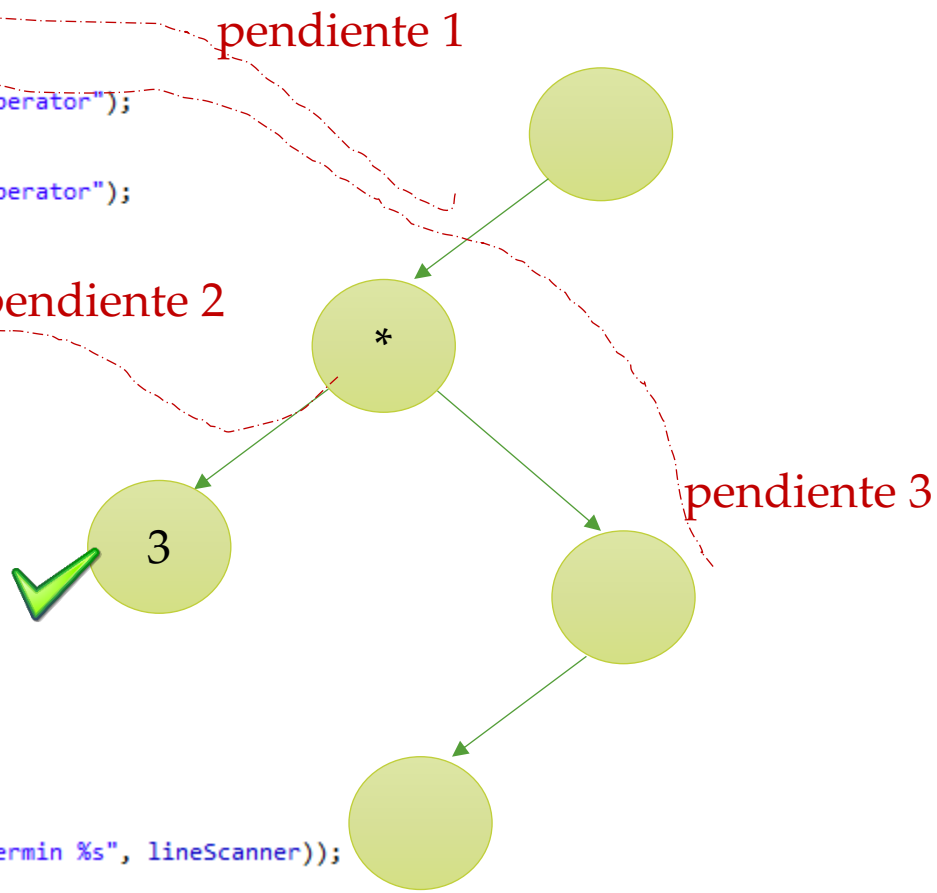
5 - 10.2)) - 2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```



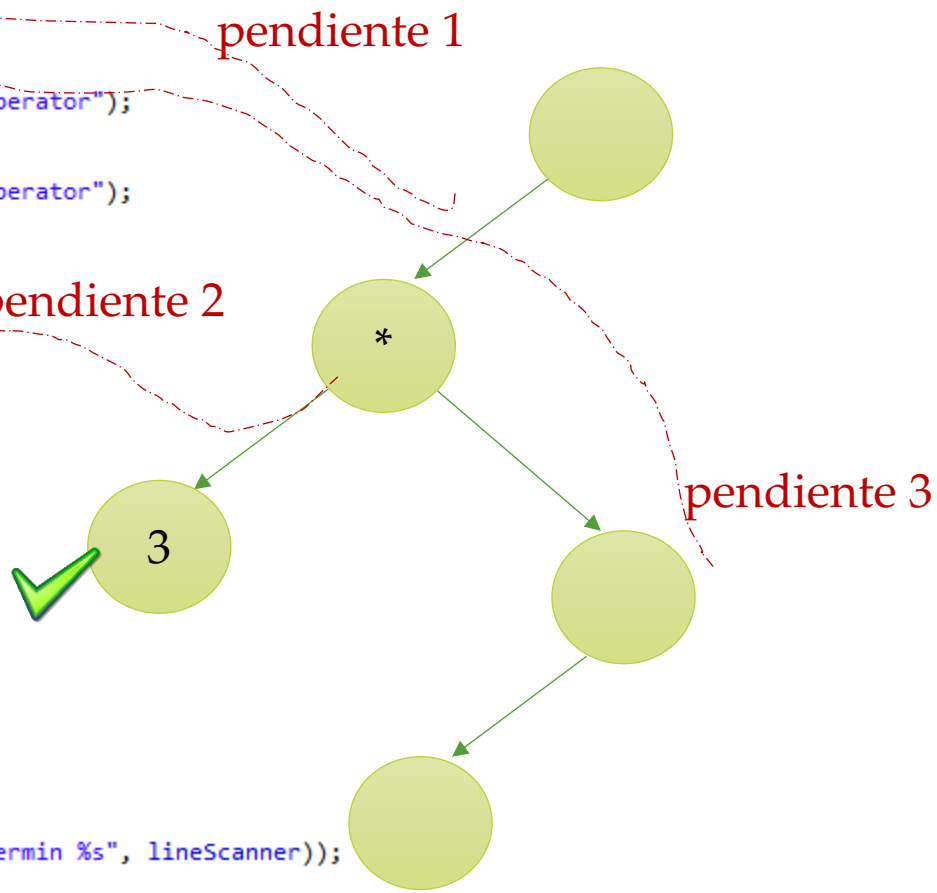
5 - 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



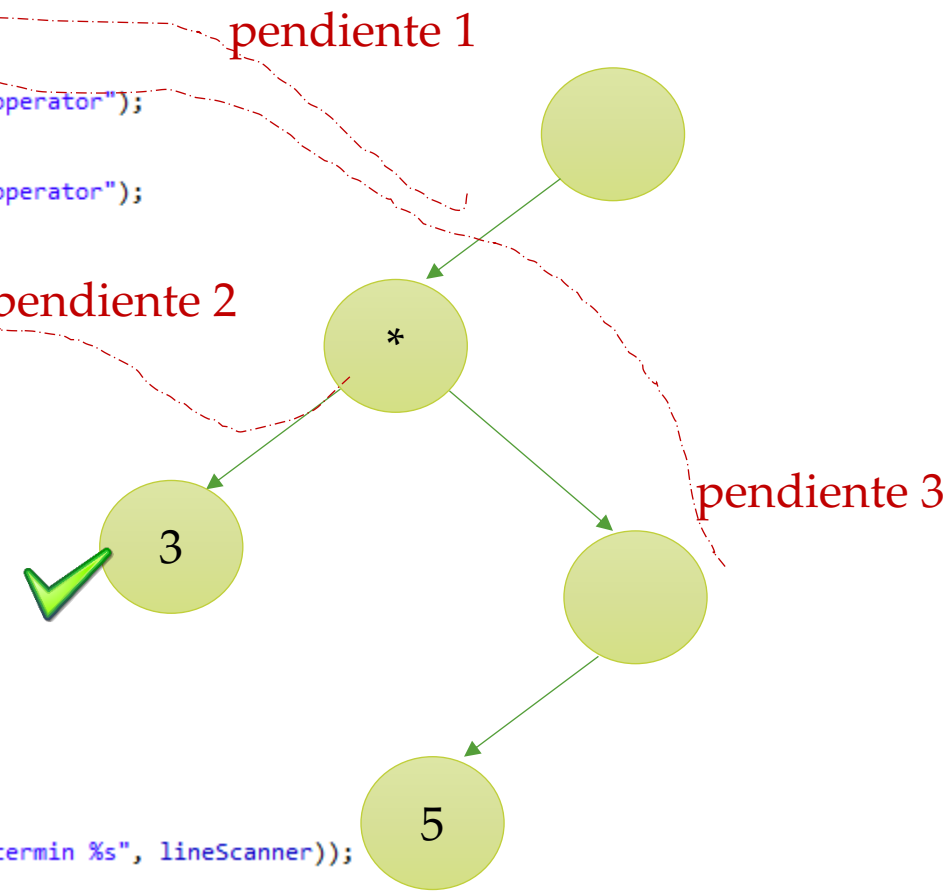
5 - 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



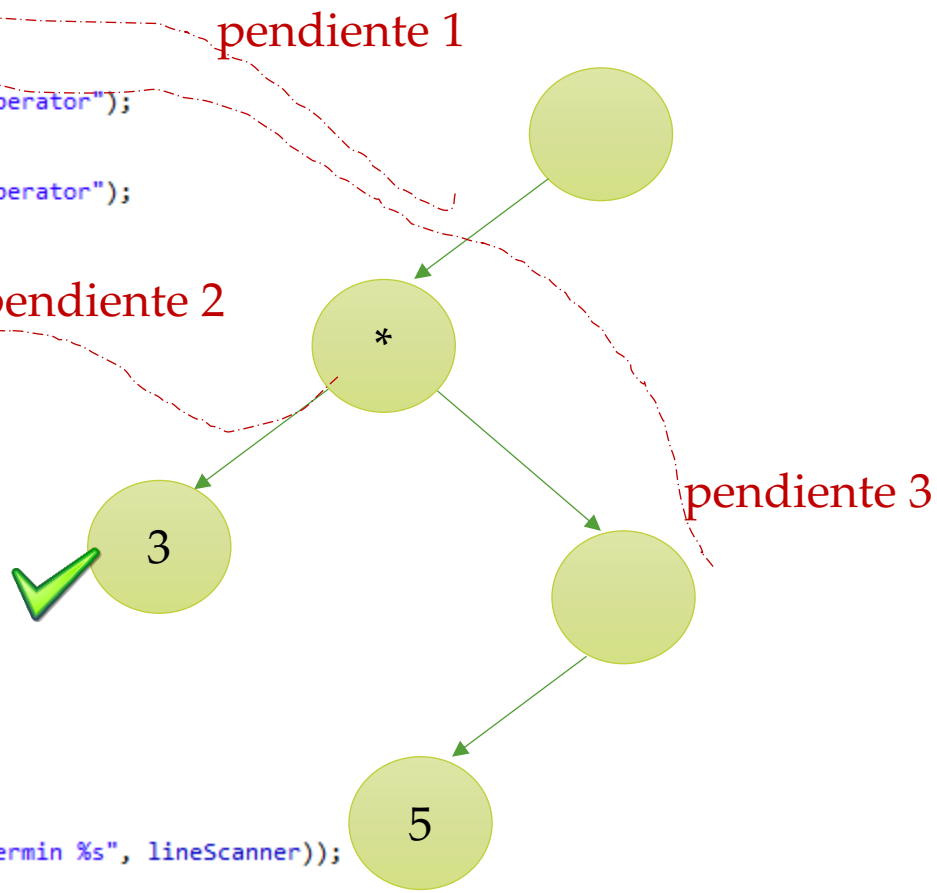
- 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



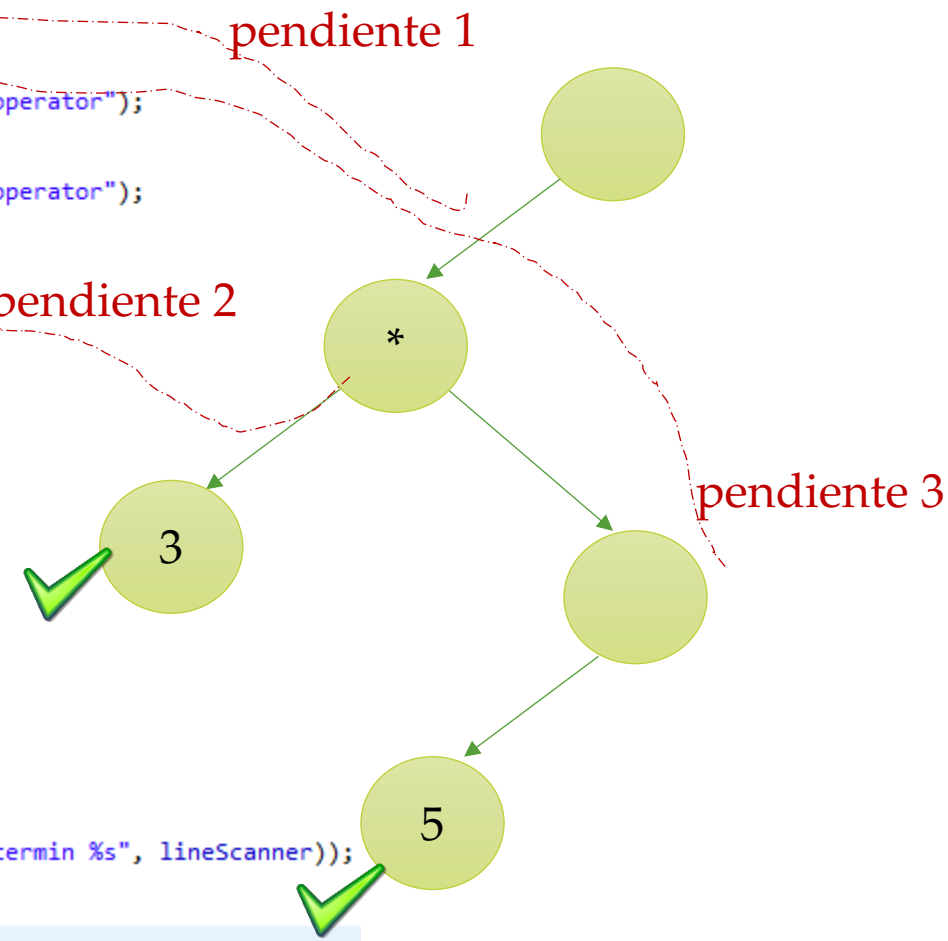
- 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



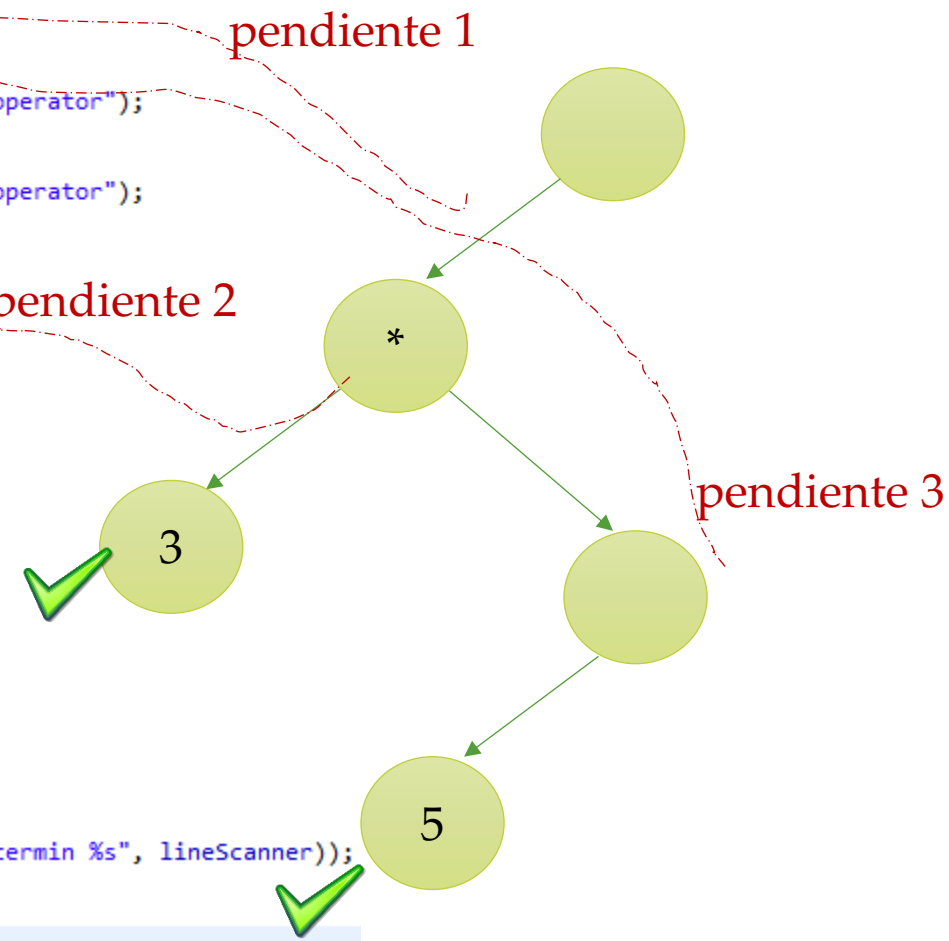
- 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



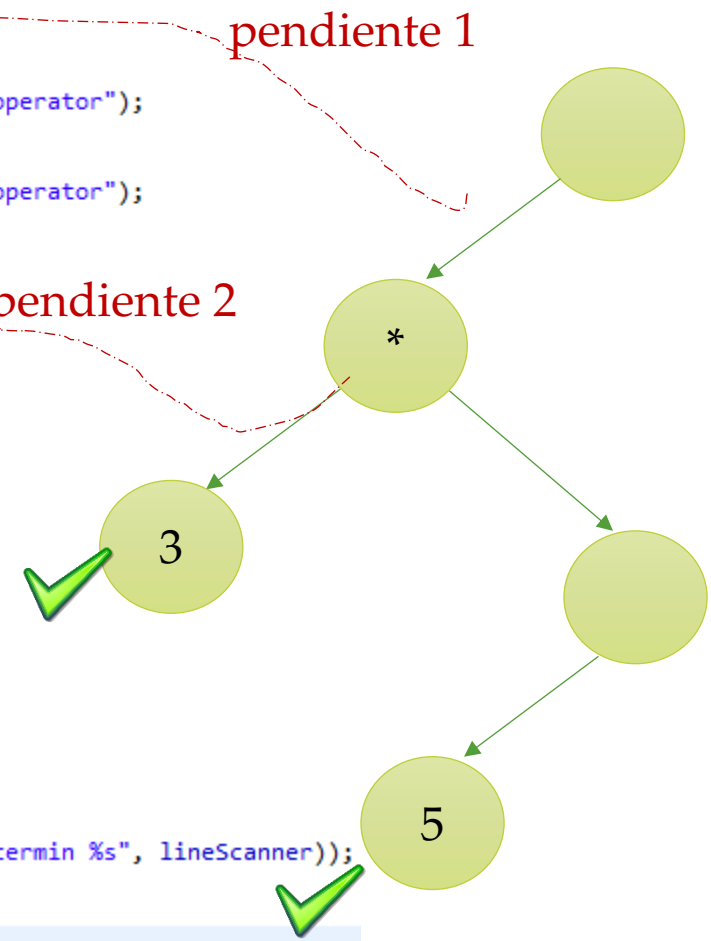
- 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



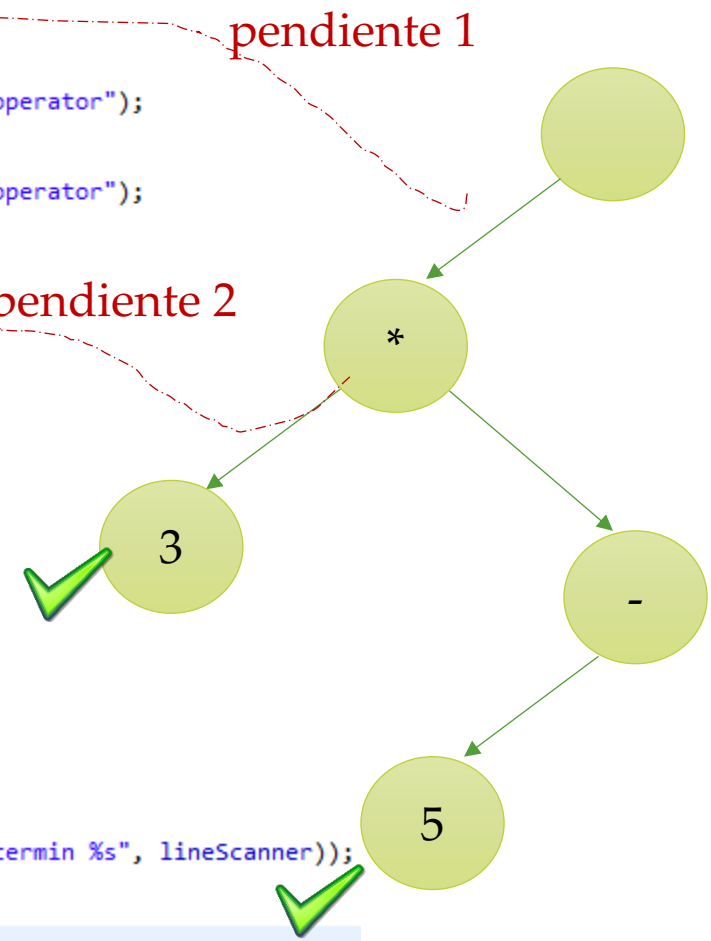
- 10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



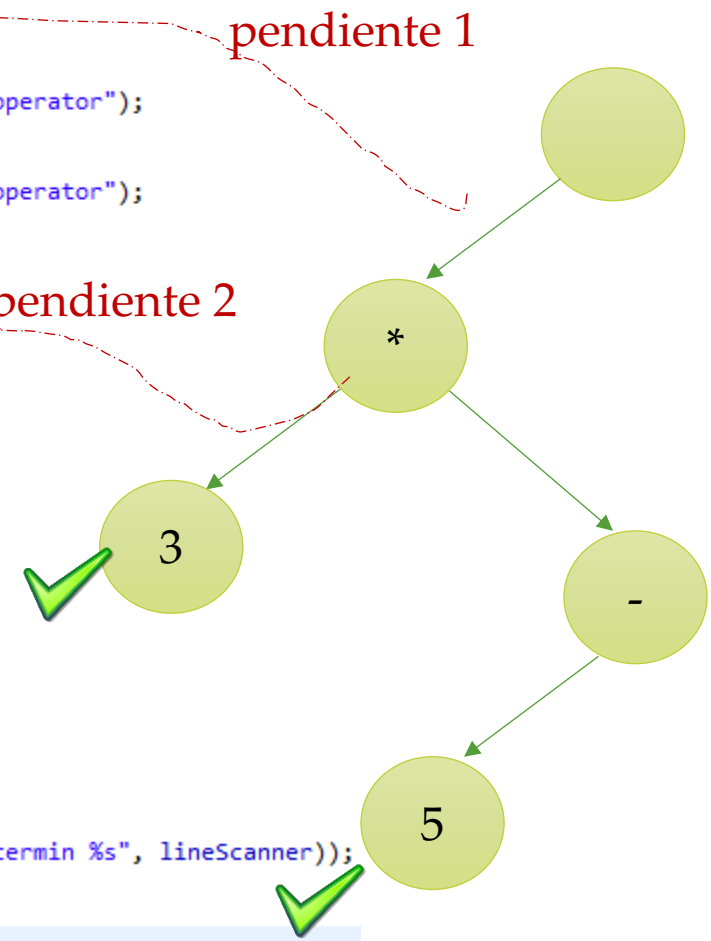
10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



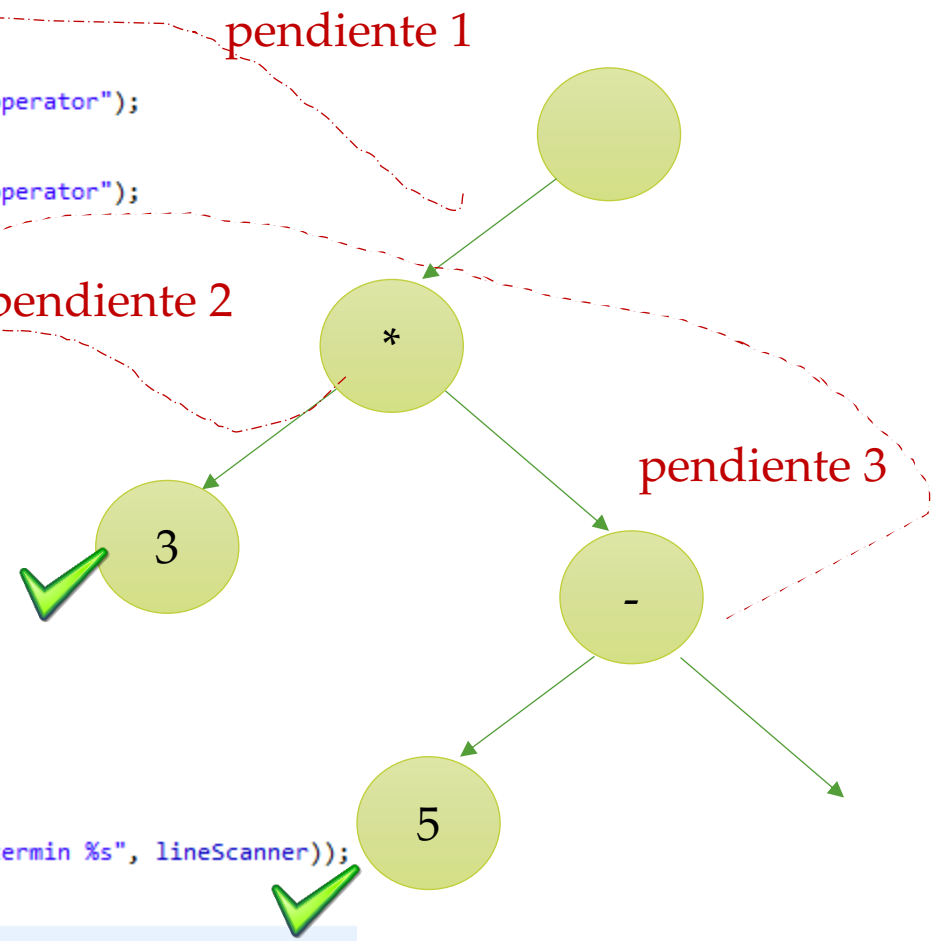
10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



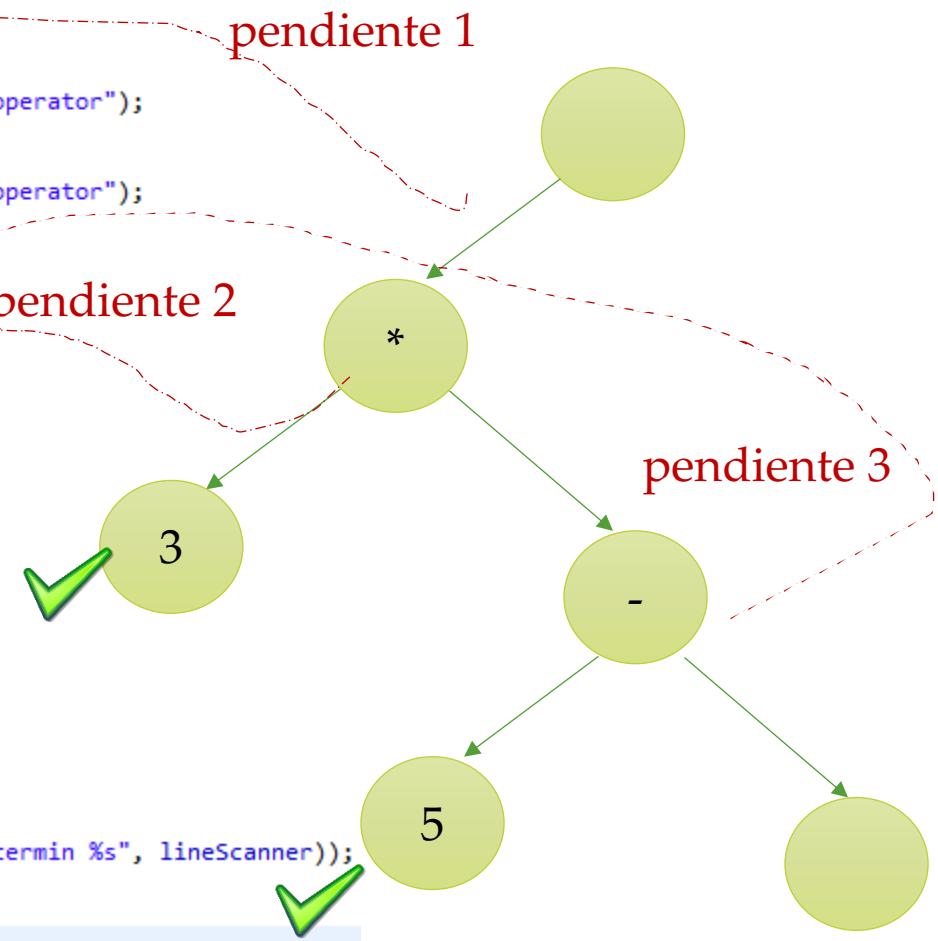
10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



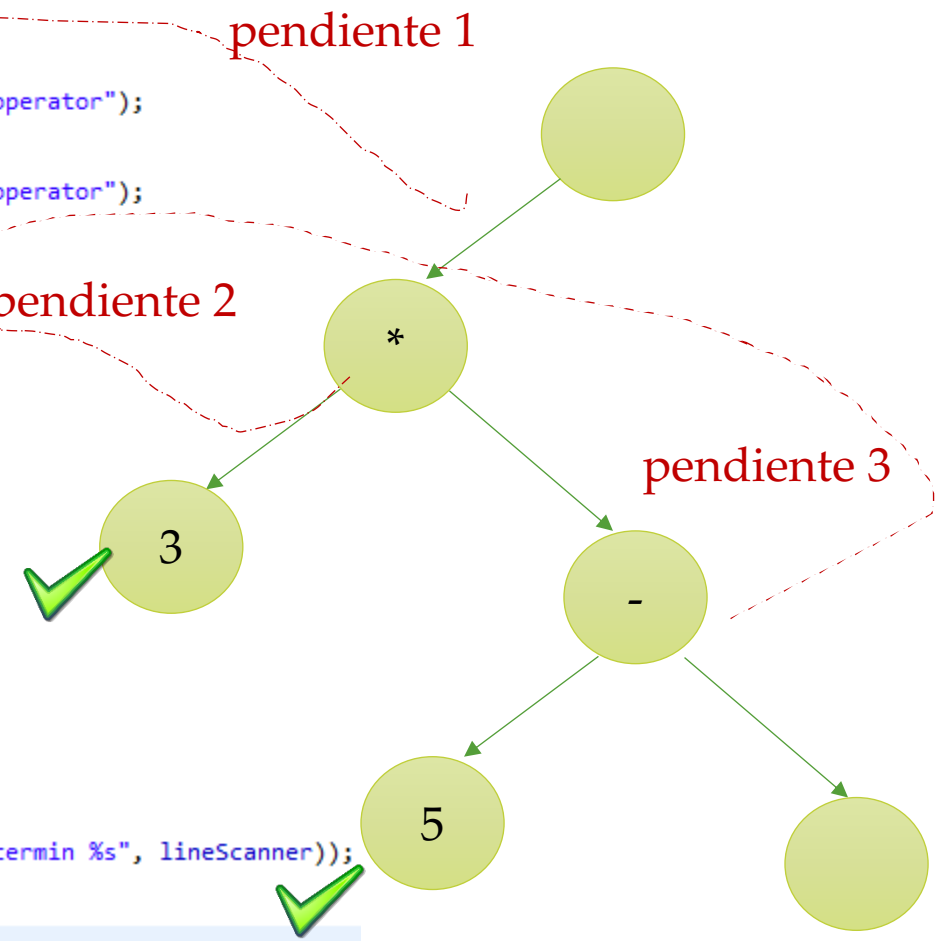
10.2)) - 2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```



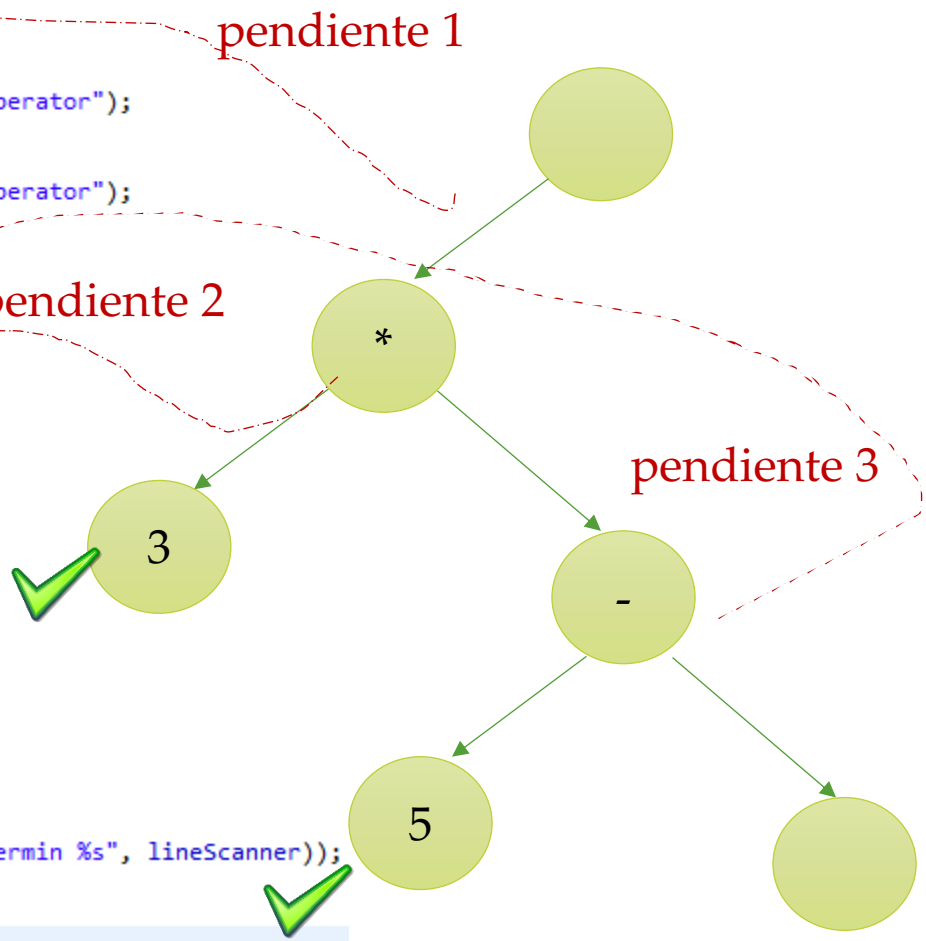
10.2)) - 2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```



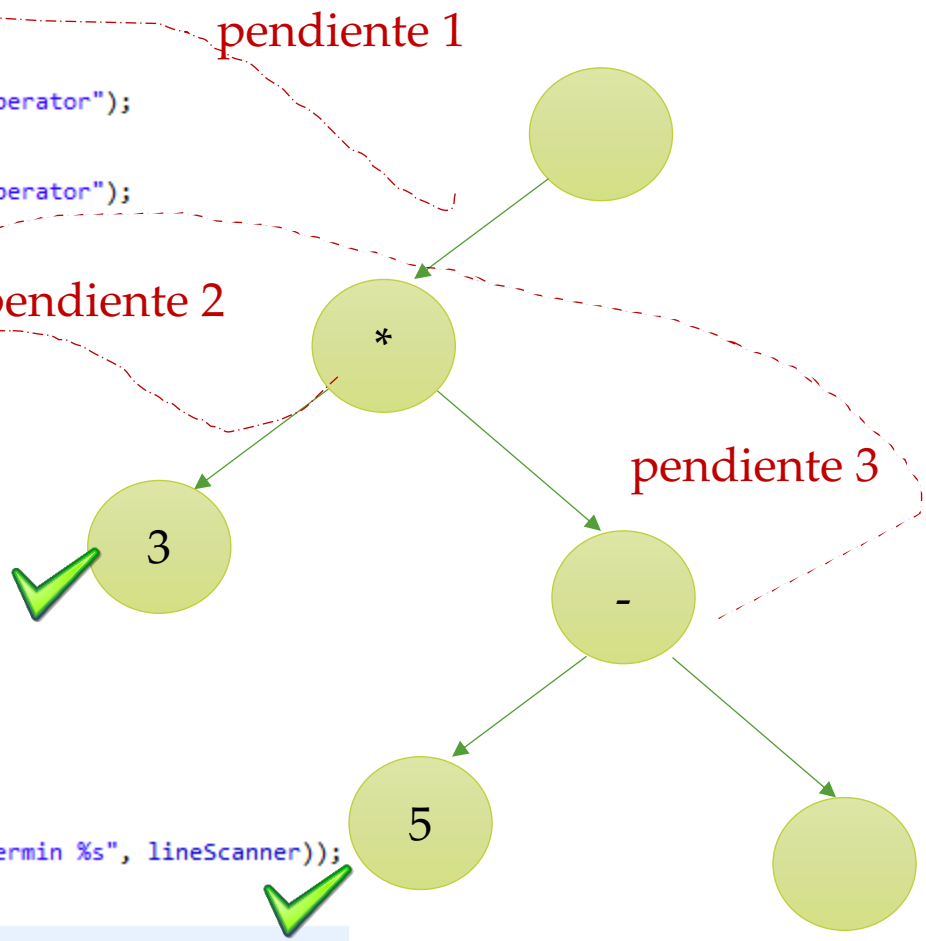
10.2)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



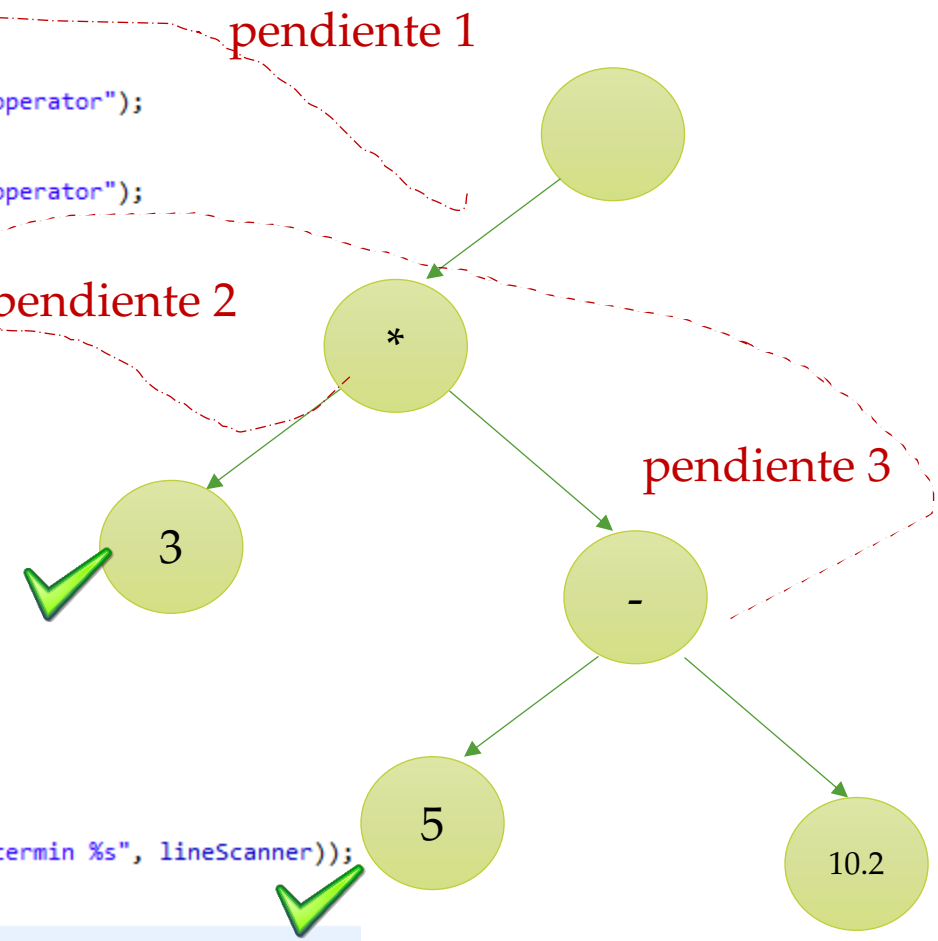
10.2)) - 2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```



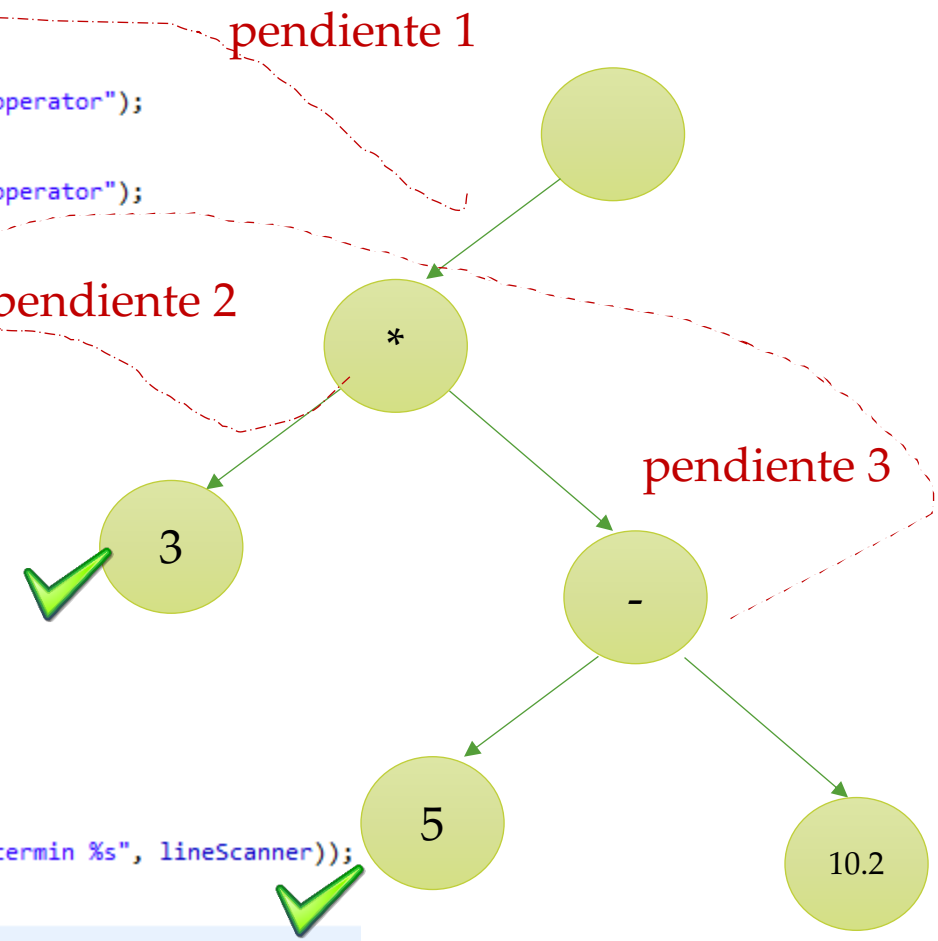
)) - 2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```



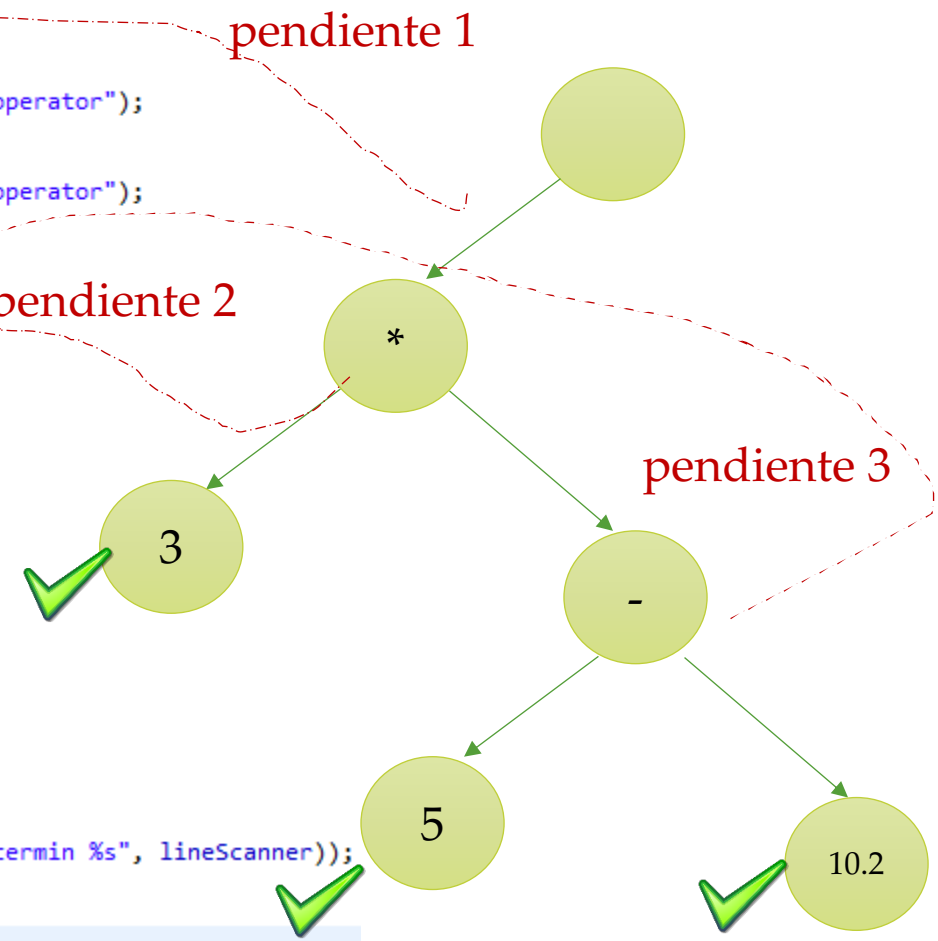
)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



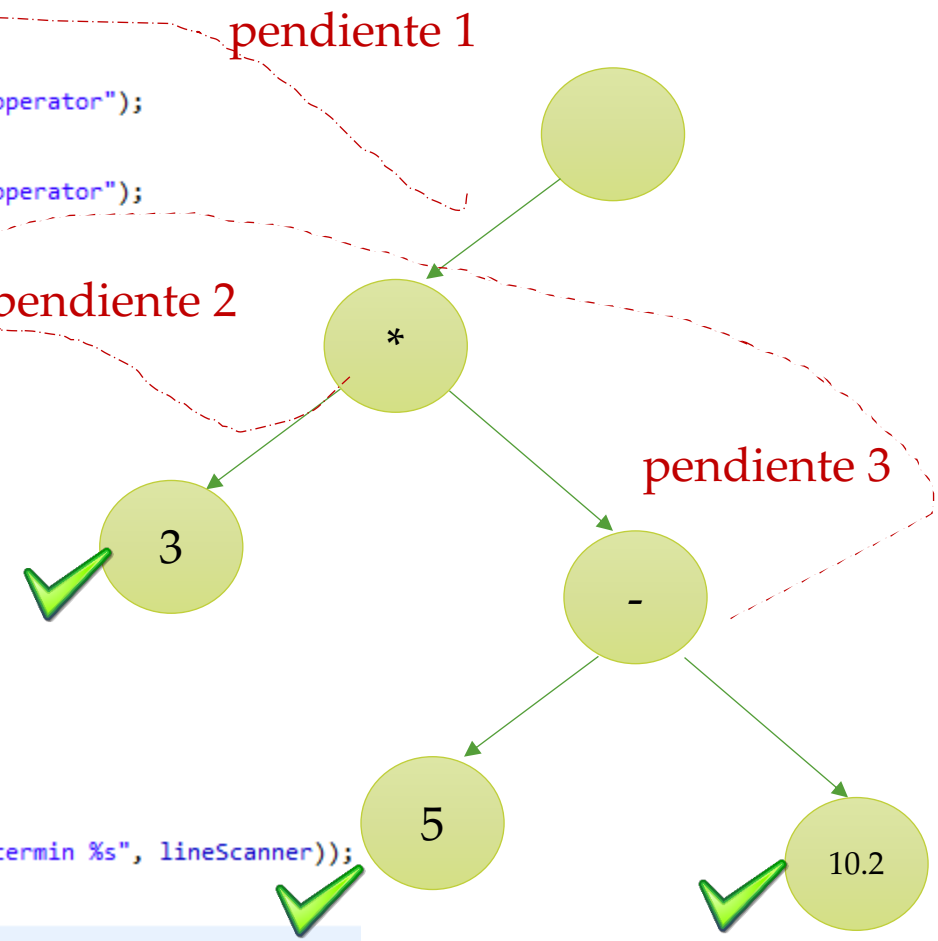
)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



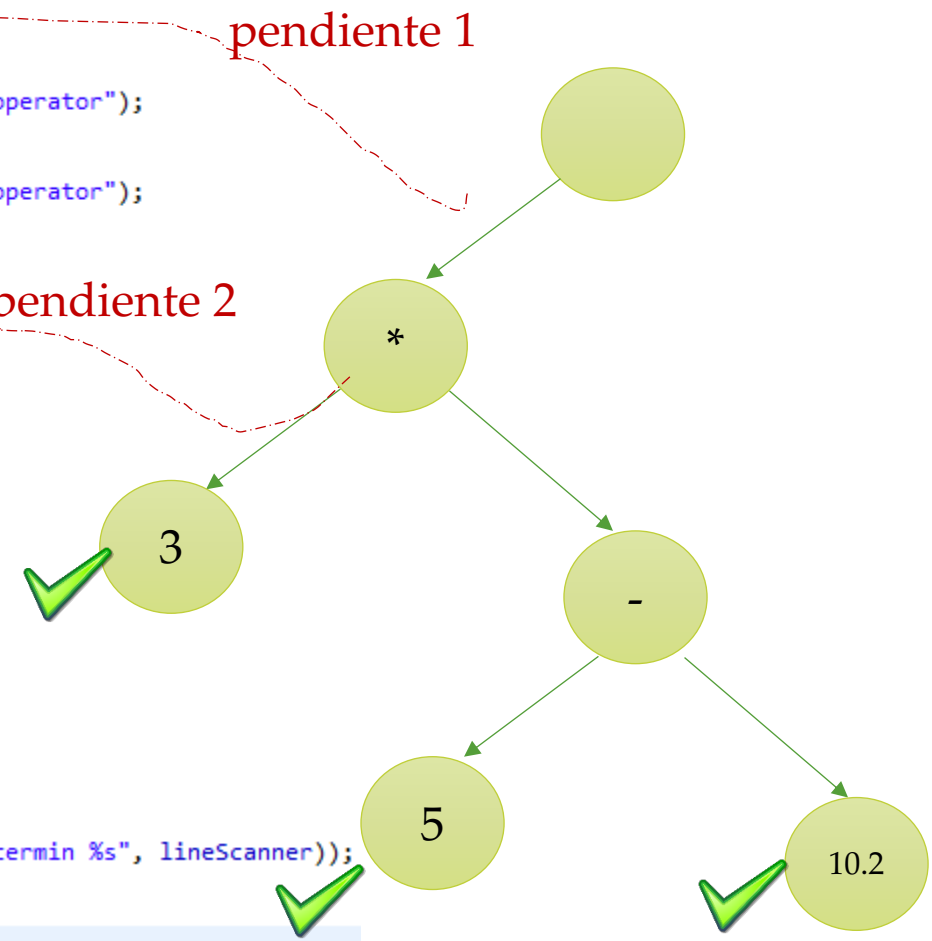
)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



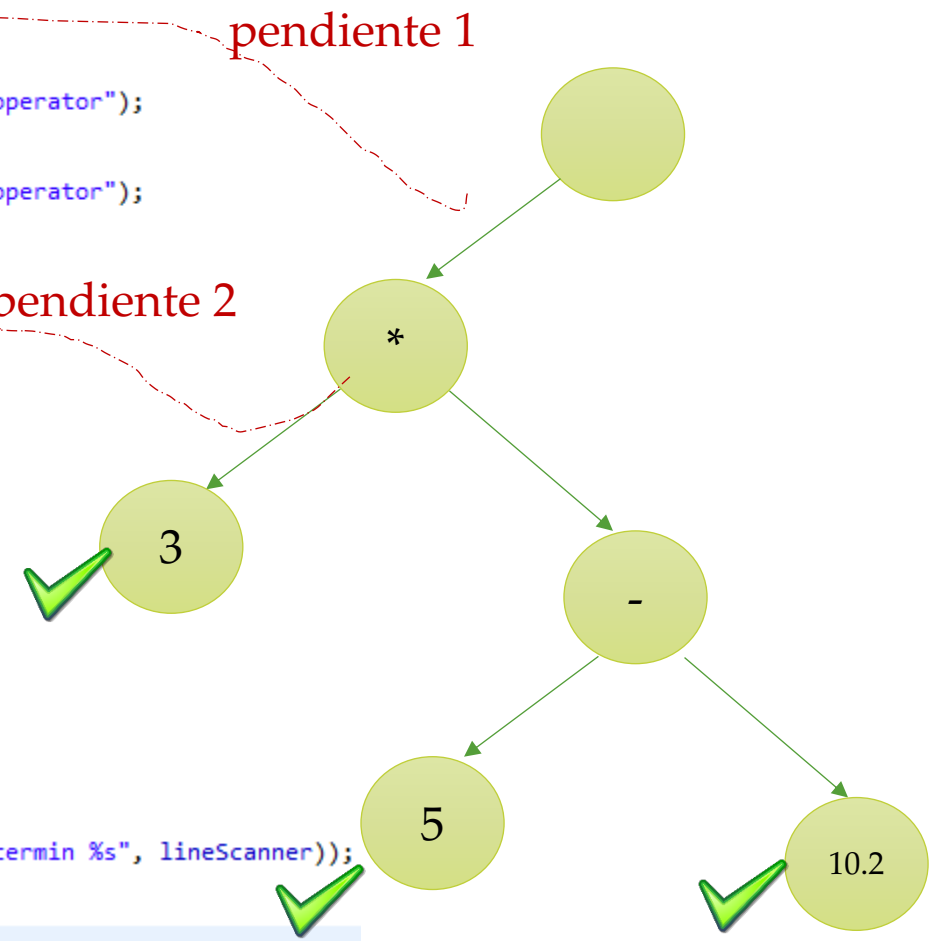
)) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



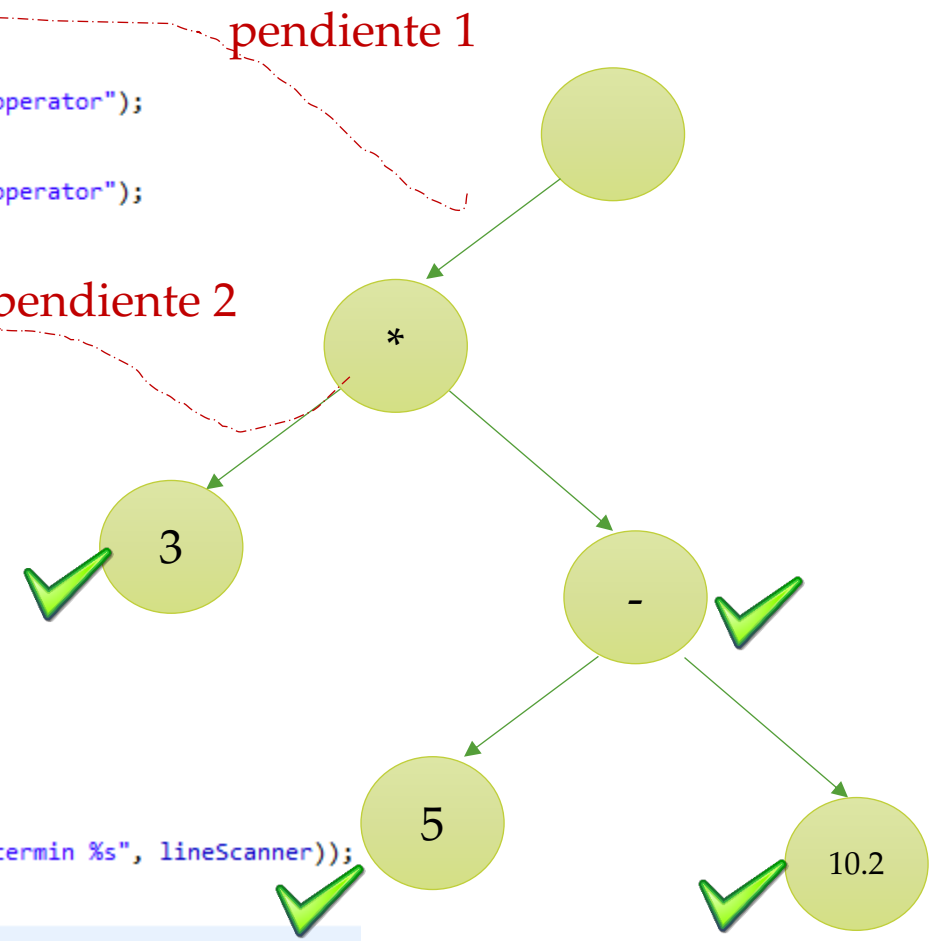
) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



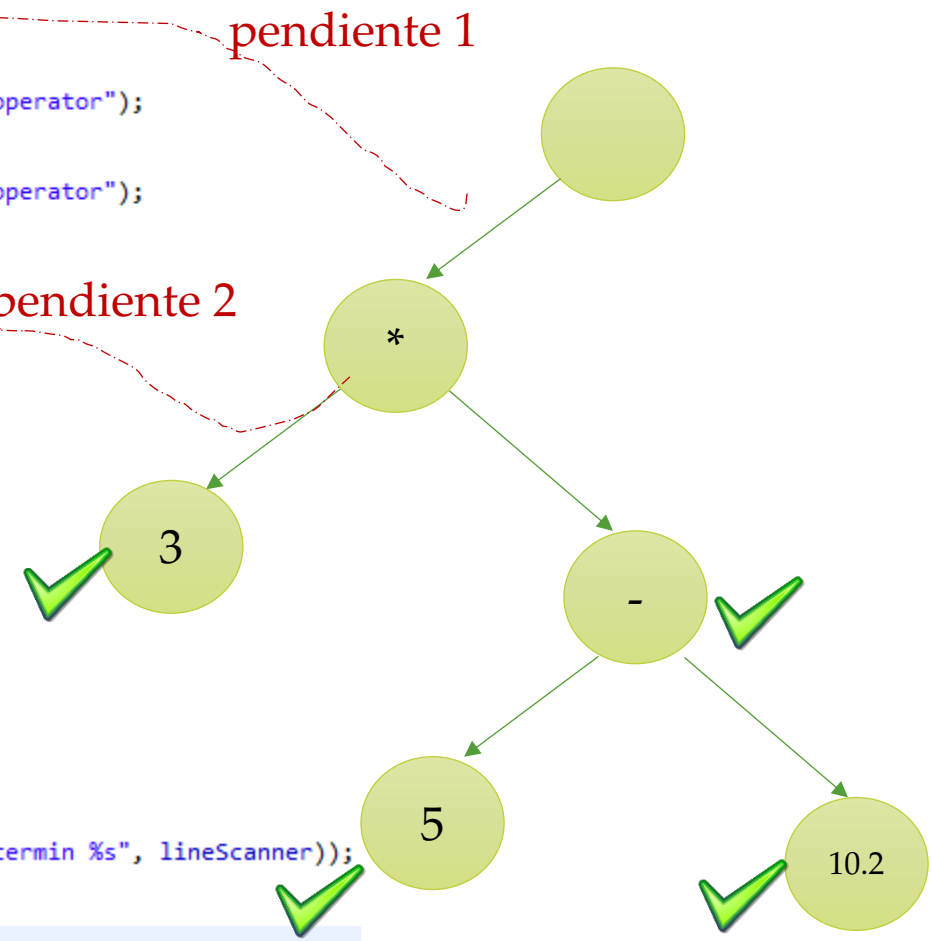
) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



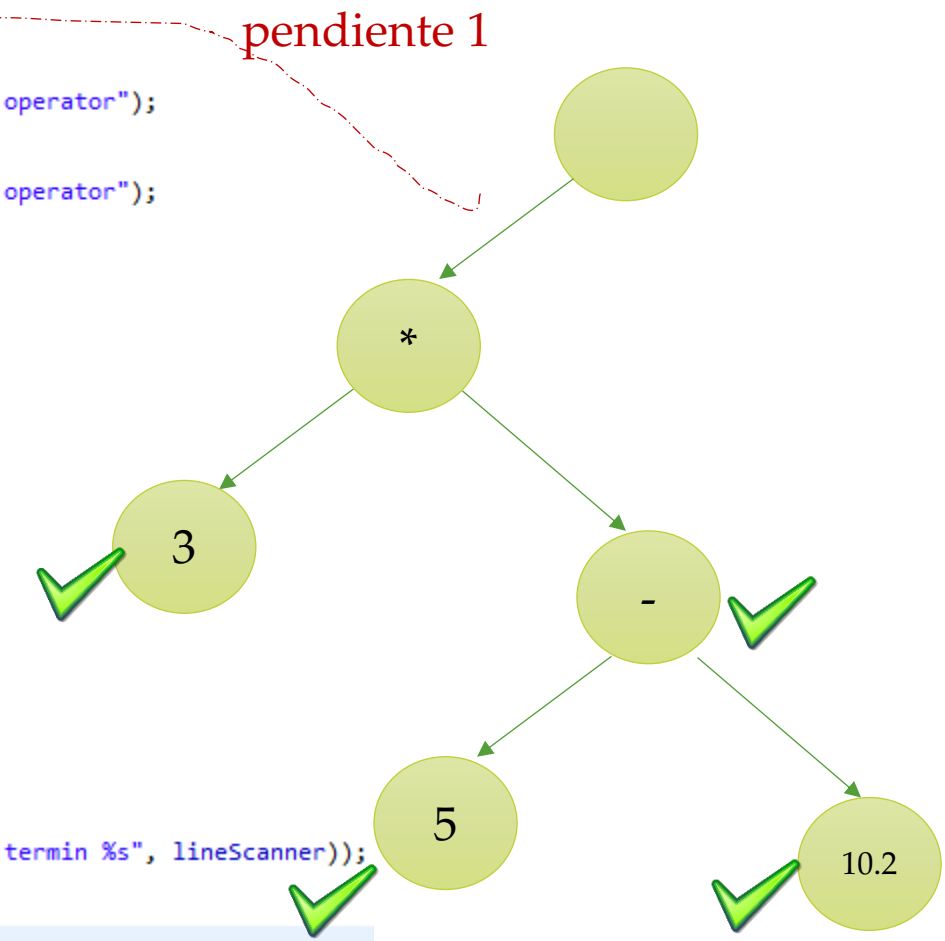
) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



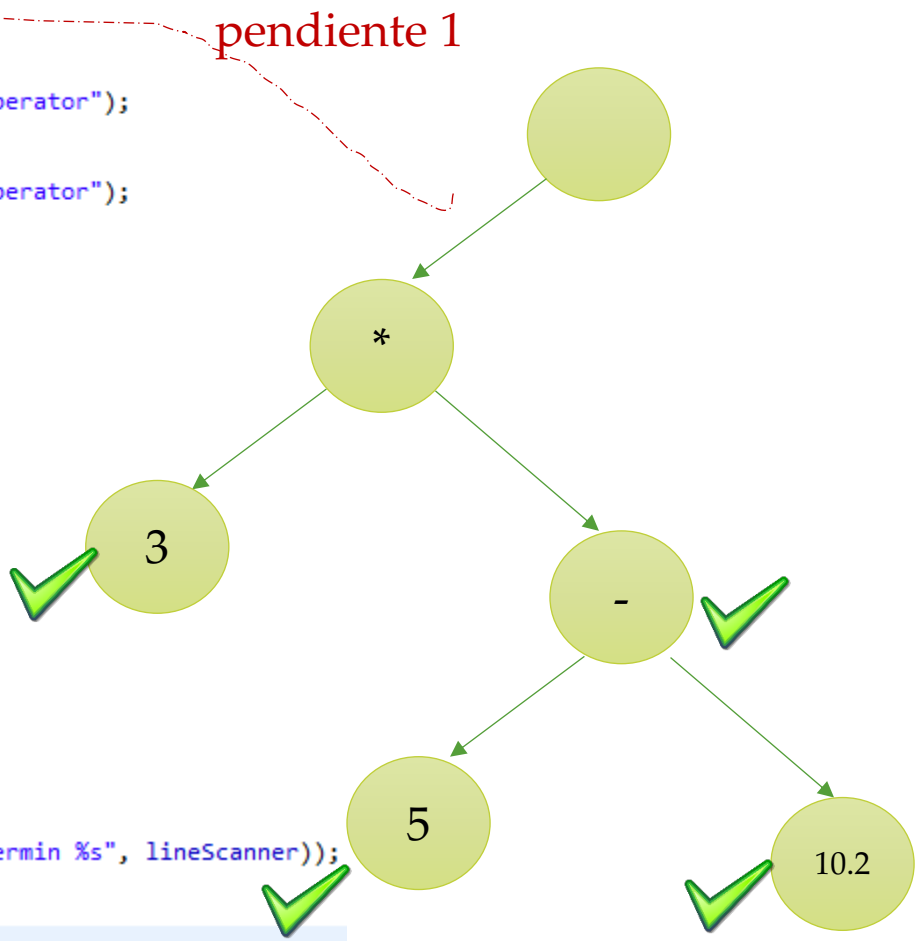
) - 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



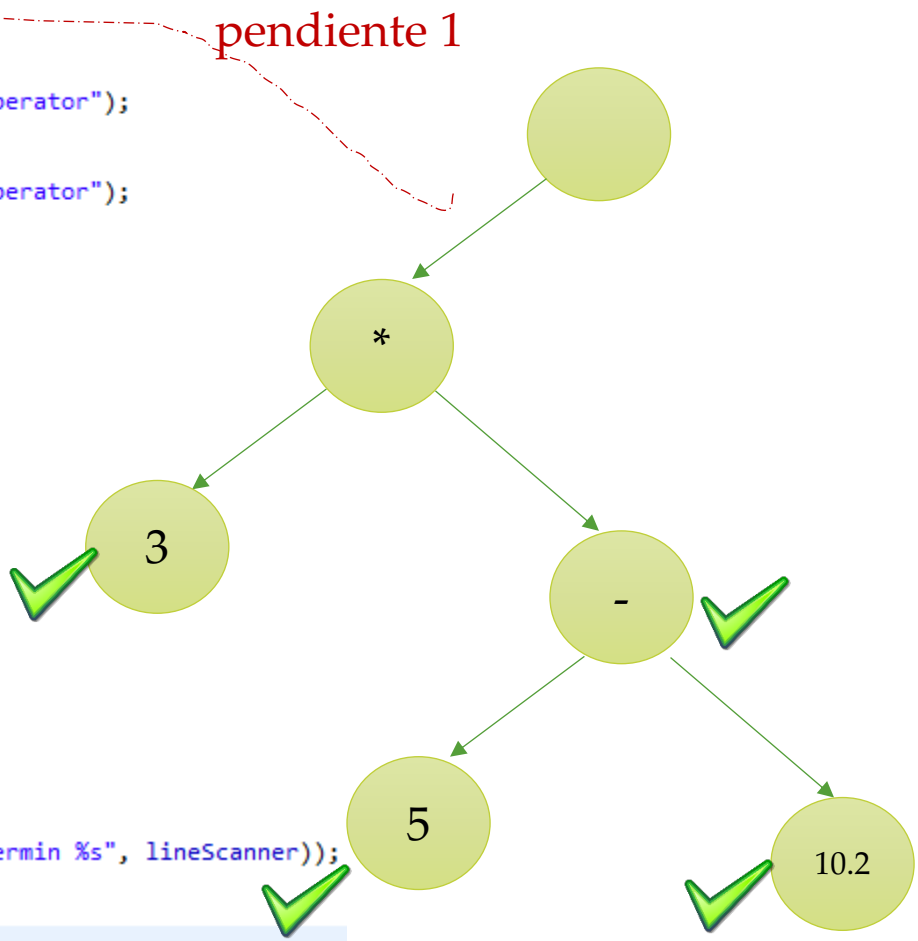
- 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



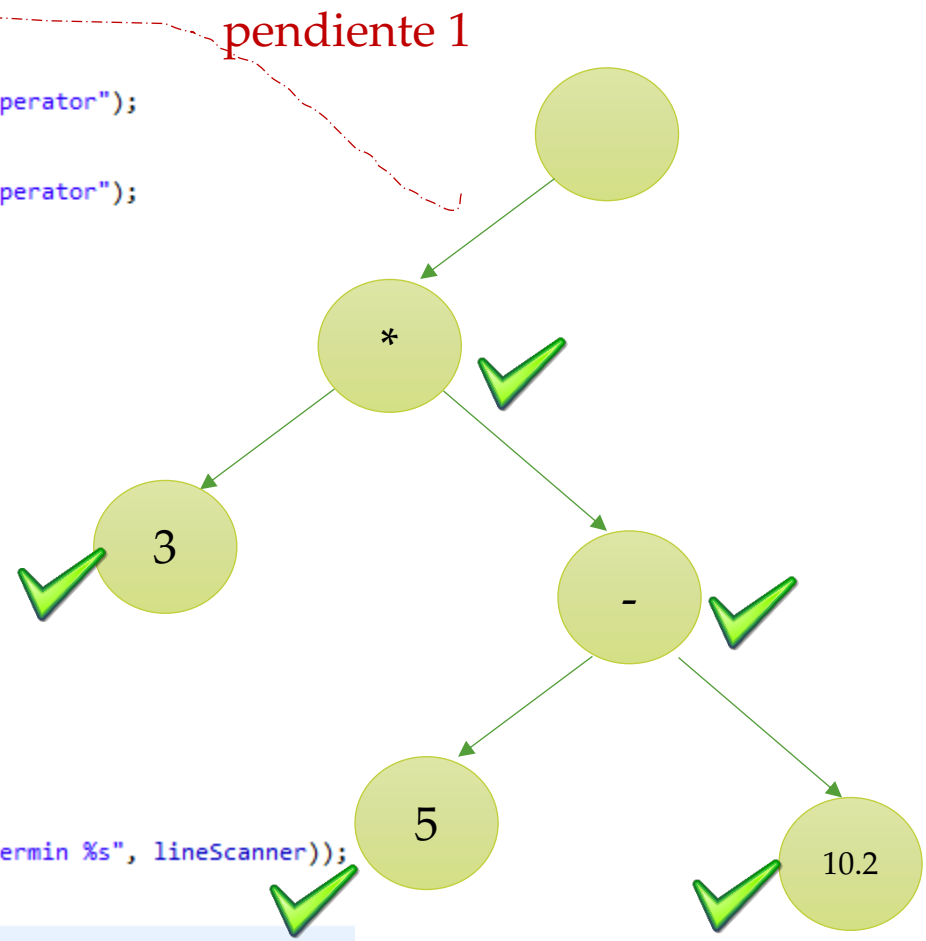
- 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



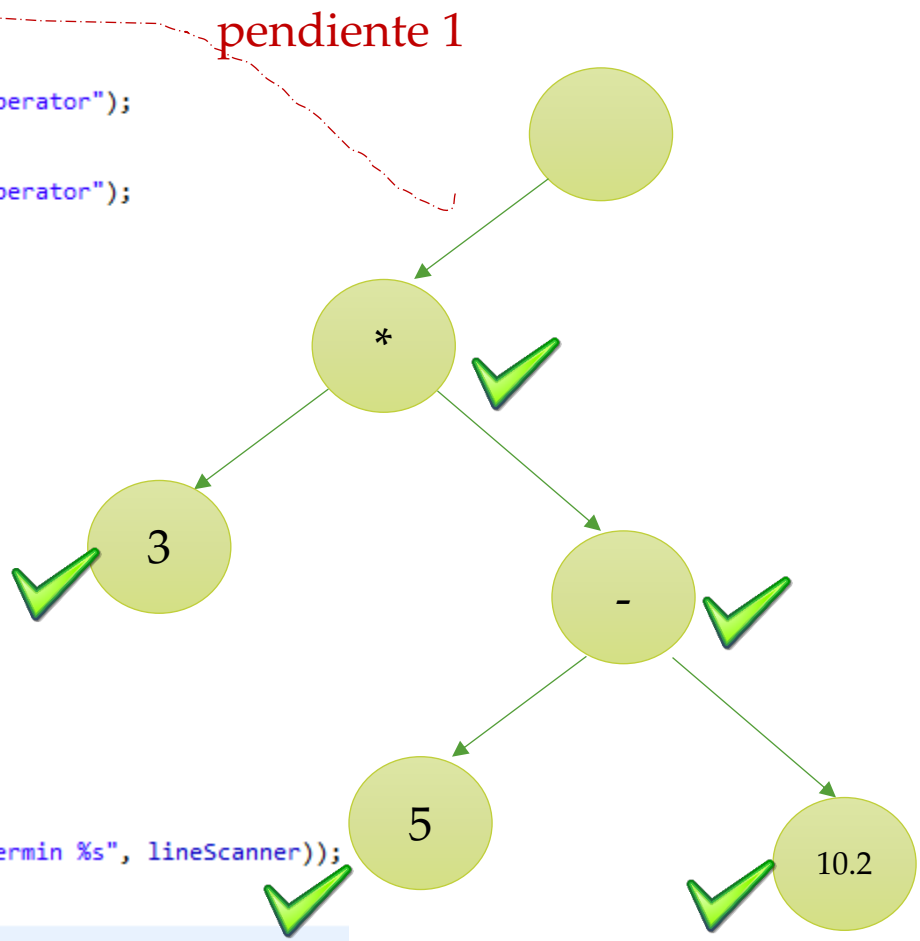
- 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



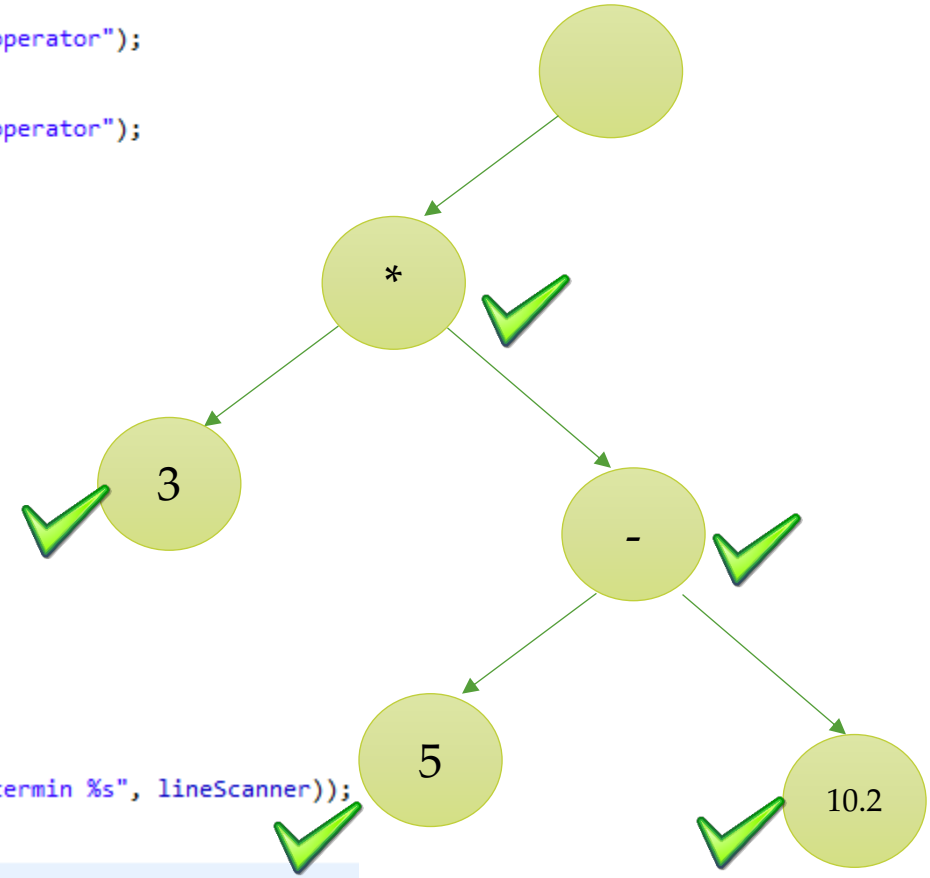
- 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



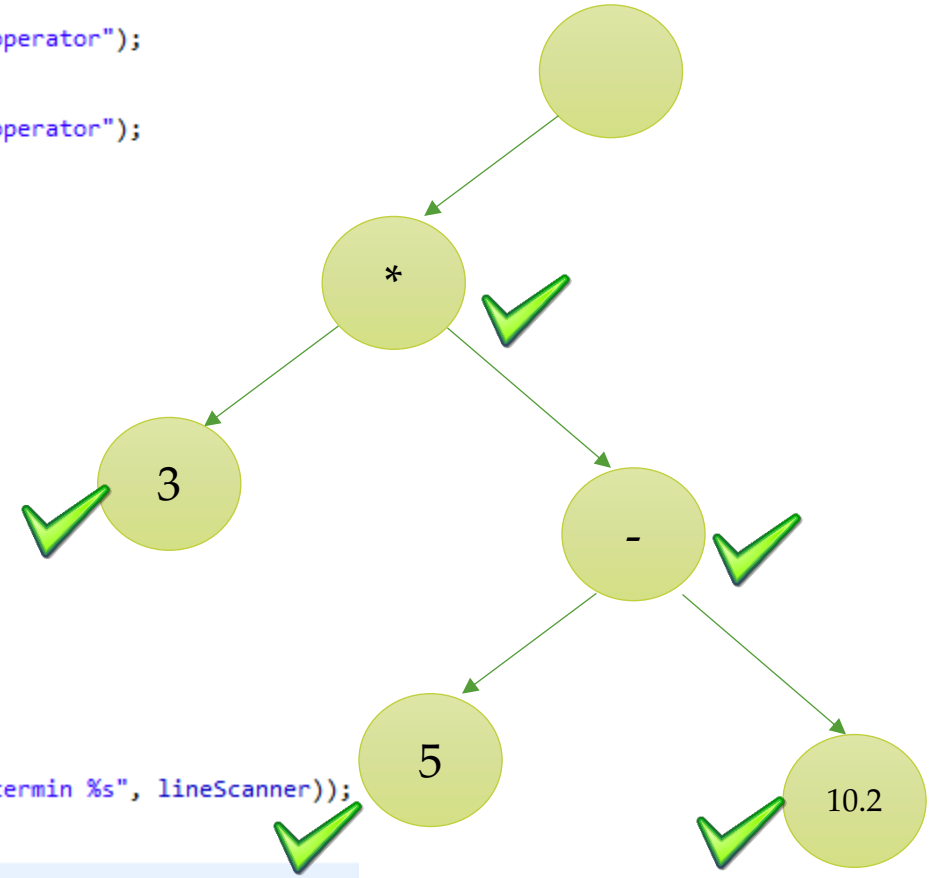
- 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



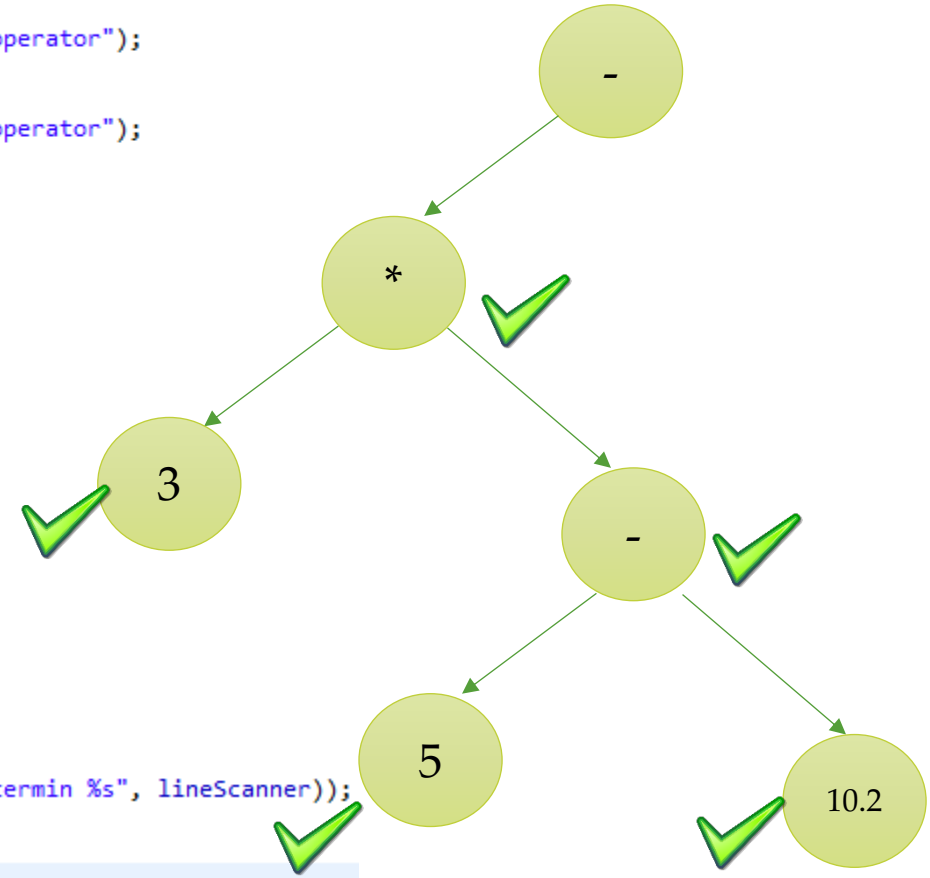
- 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



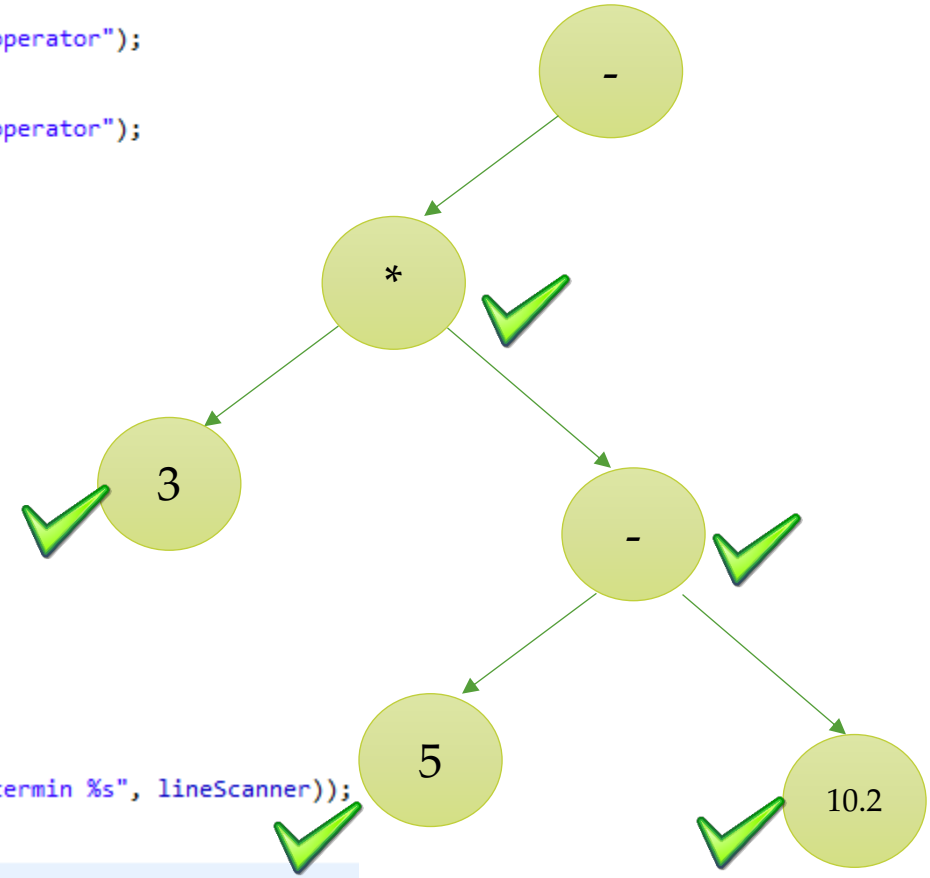
- 2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



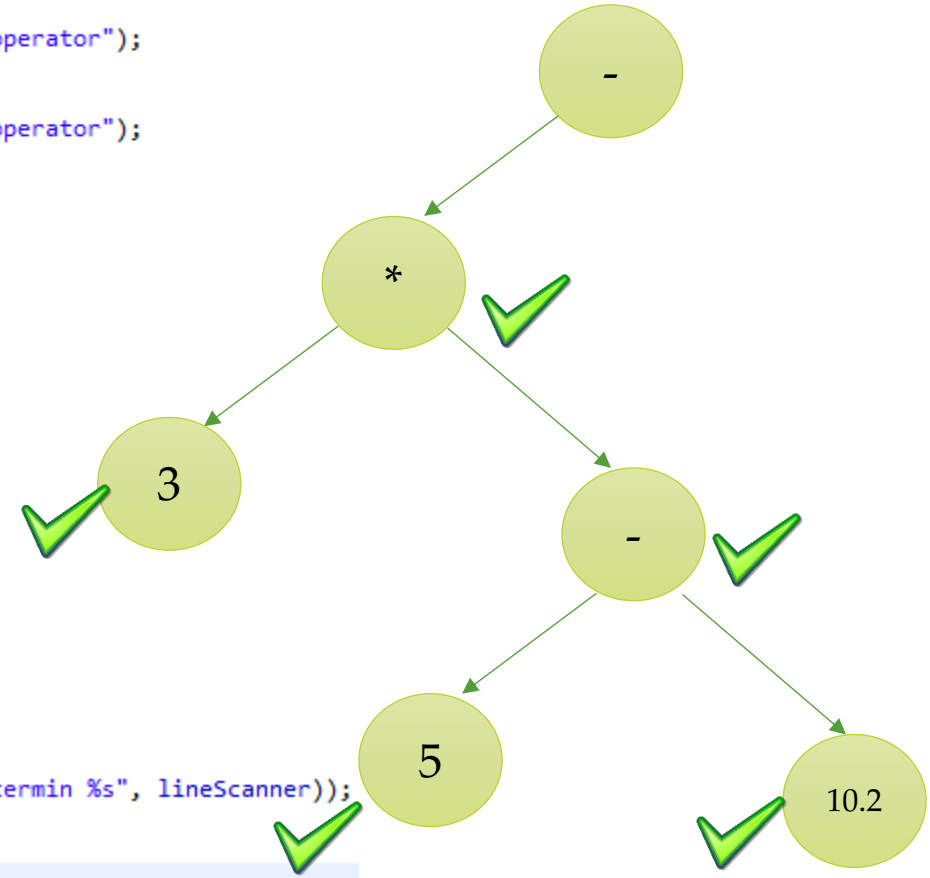
2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



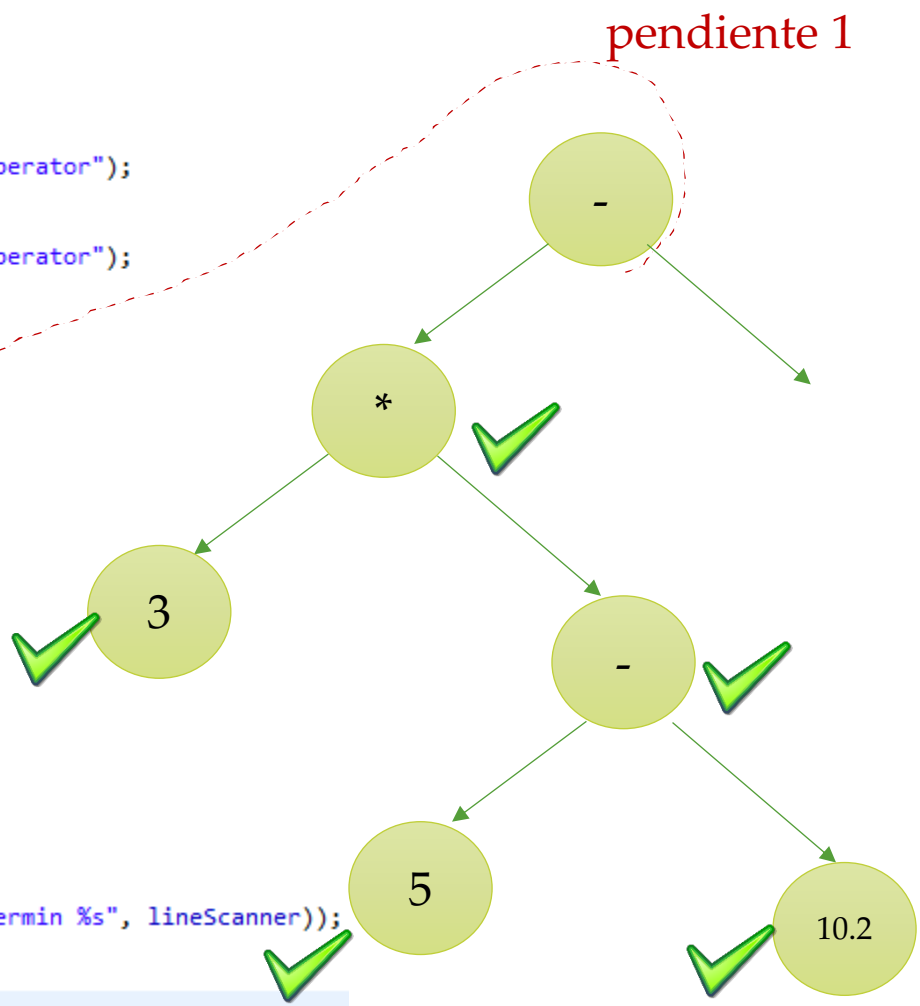
2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



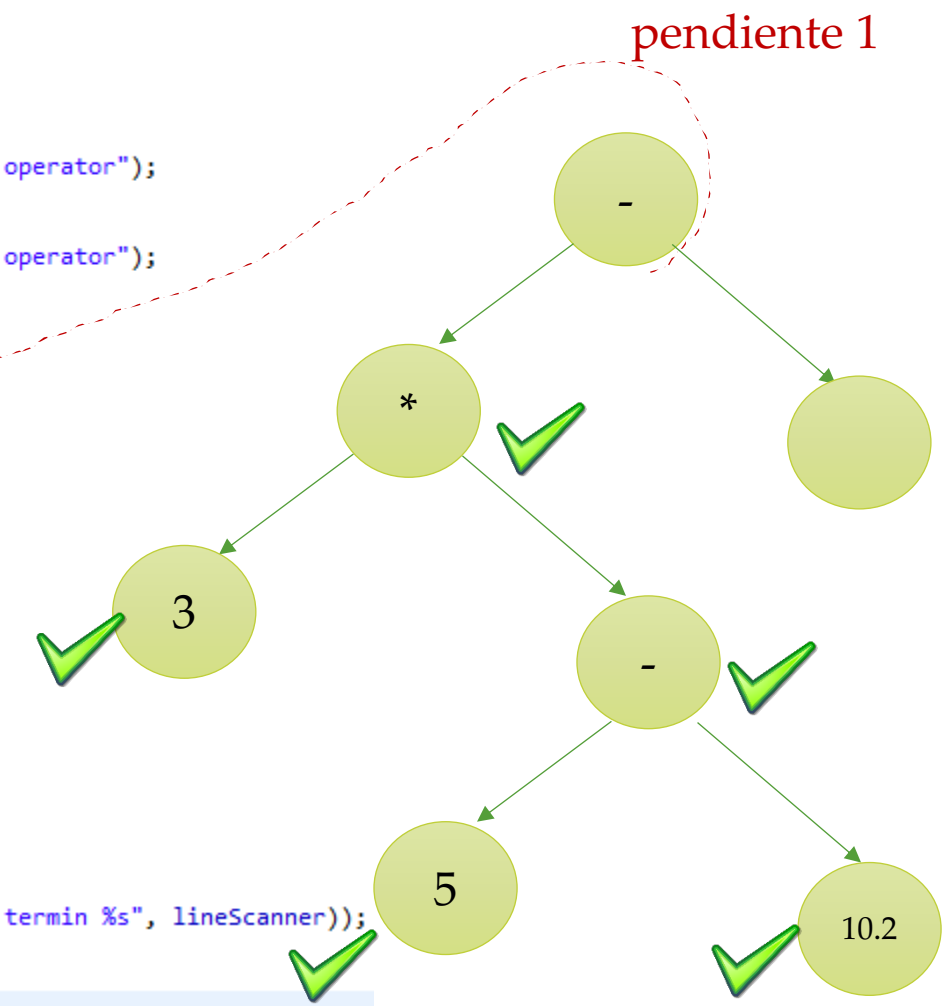
2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



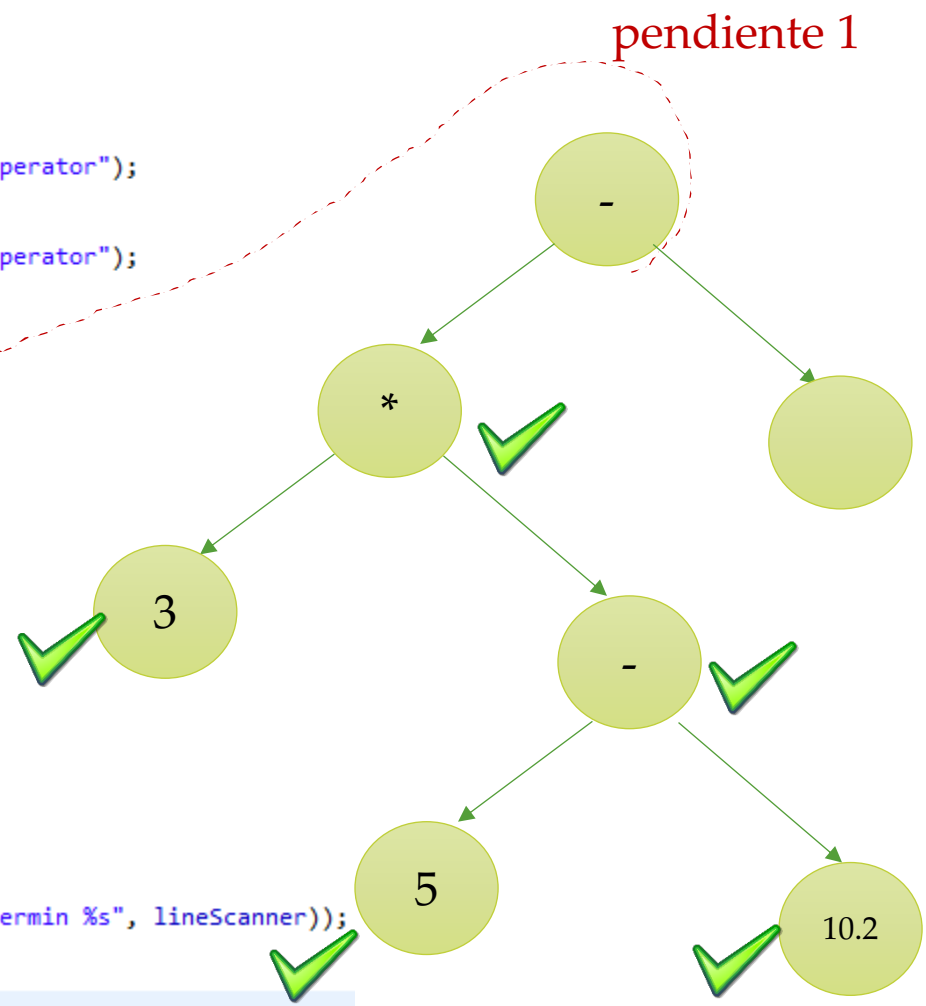
2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```



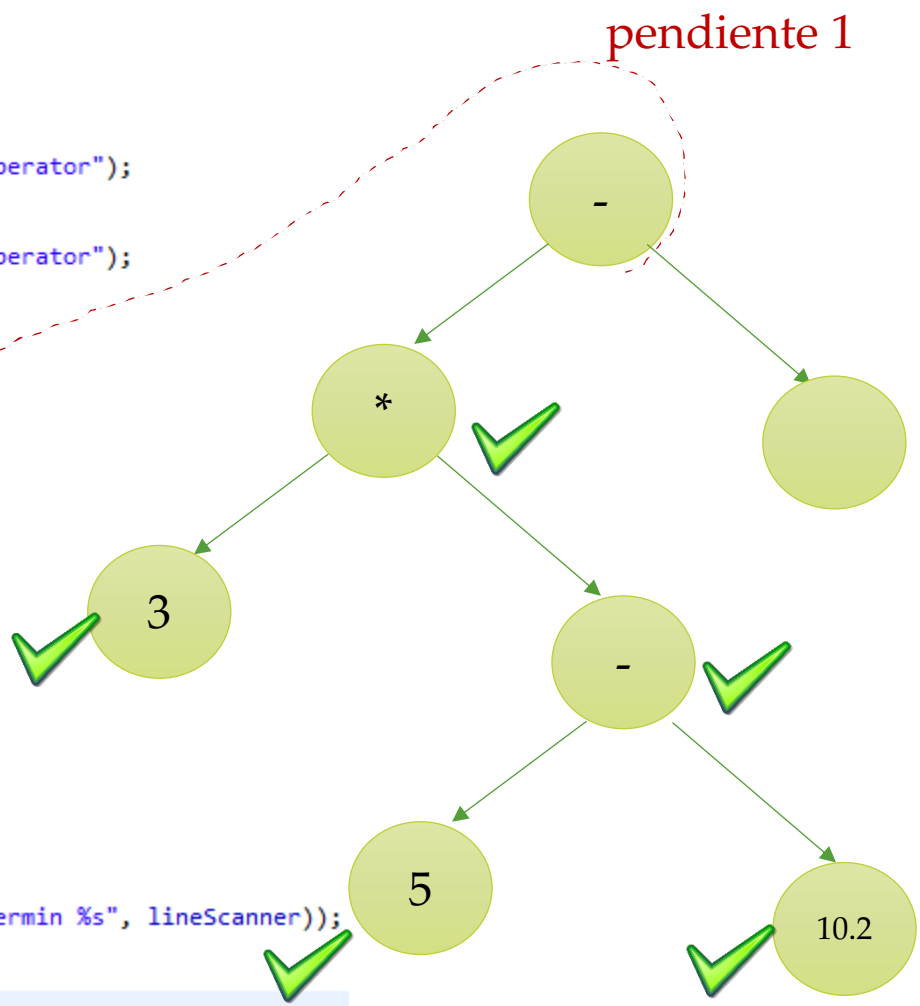
2)

```java
private Node buildExpression() {
    Node n = new Node();

    if (lineScanner.hasNext("\\(")) {
        lineScanner.next(); // lo consumo

        n.left = buildExpression(); // subexpression

        // operator
        if (!lineScanner.hasNext())
            throw new RuntimeException("missing or invalid operator");
        n.data = lineScanner.next();
        if (!Utils.isOperator(n.data))
            throw new RuntimeException("missing or invalid operator");

        // subexpression
        n.right = buildExpression();

        // ) expected
        if (lineScanner.hasNext("\\)")) {
            // lo consumo
            lineScanner.next();
        } else {
            throw new RuntimeException("missing )" );
        }

        return n;
    }

    // constant
    if (!lineScanner.hasNext())
        throw new RuntimeException("missing expression");

    n.data = lineScanner.next();
    if (!Utils.isConstant(n.data)) {
        throw new RuntimeException(String.format("illegal termin %s", lineScanner));
    }
    return n;
}
```



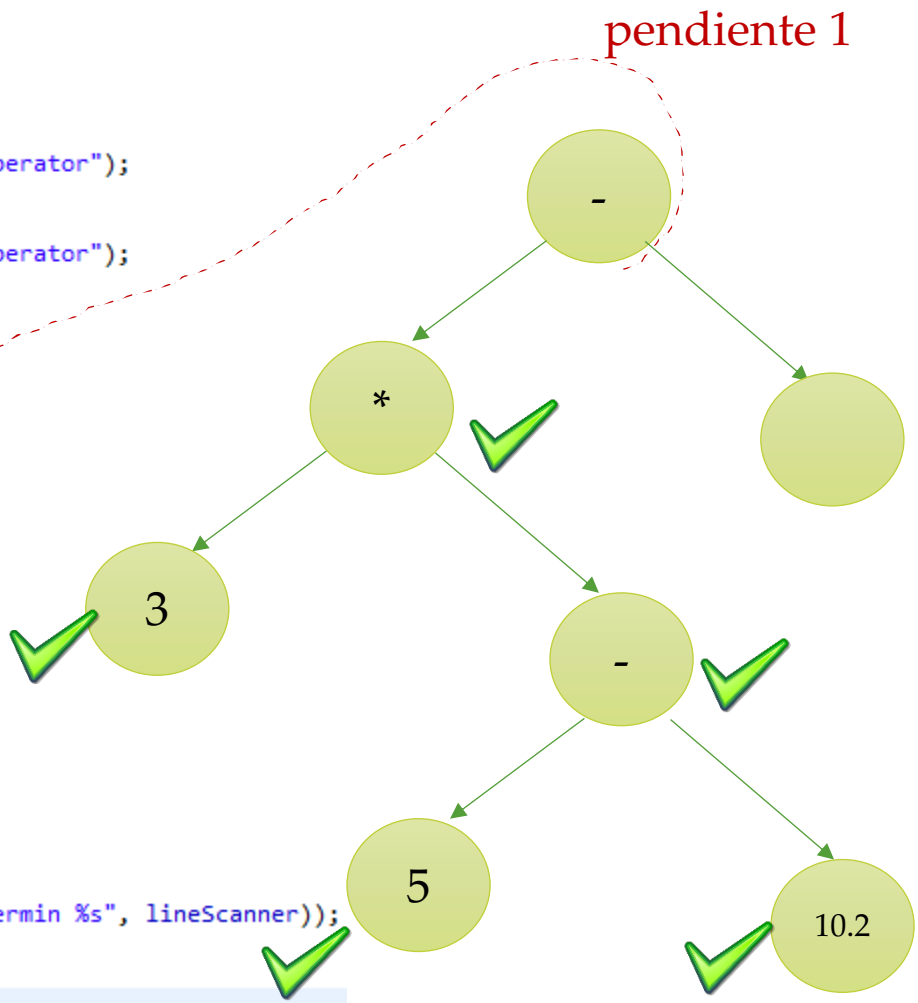
2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



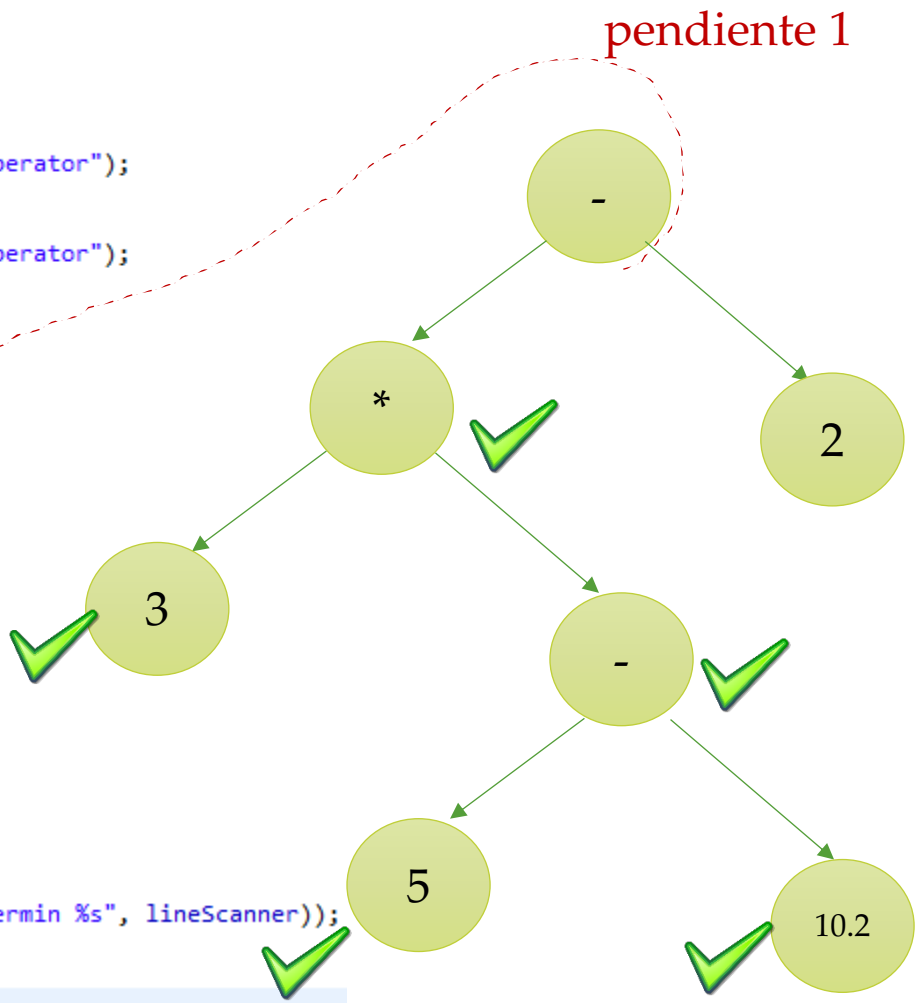
2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



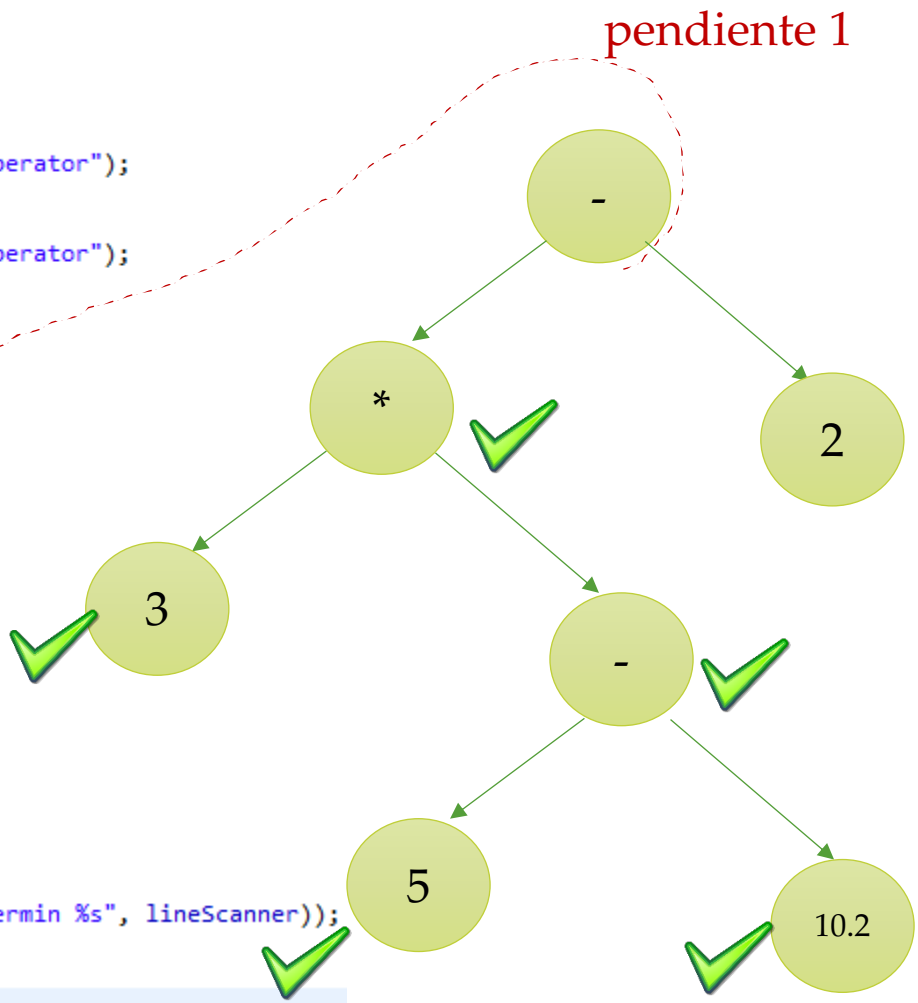
2)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



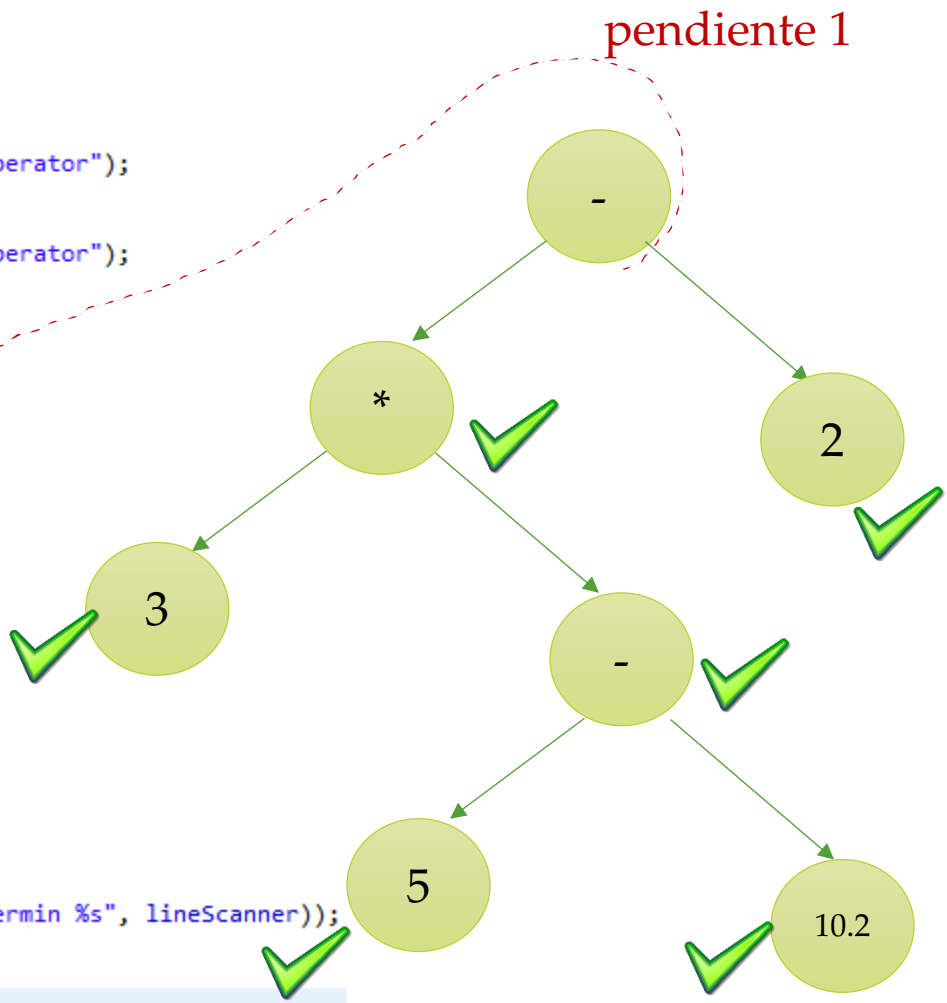
)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



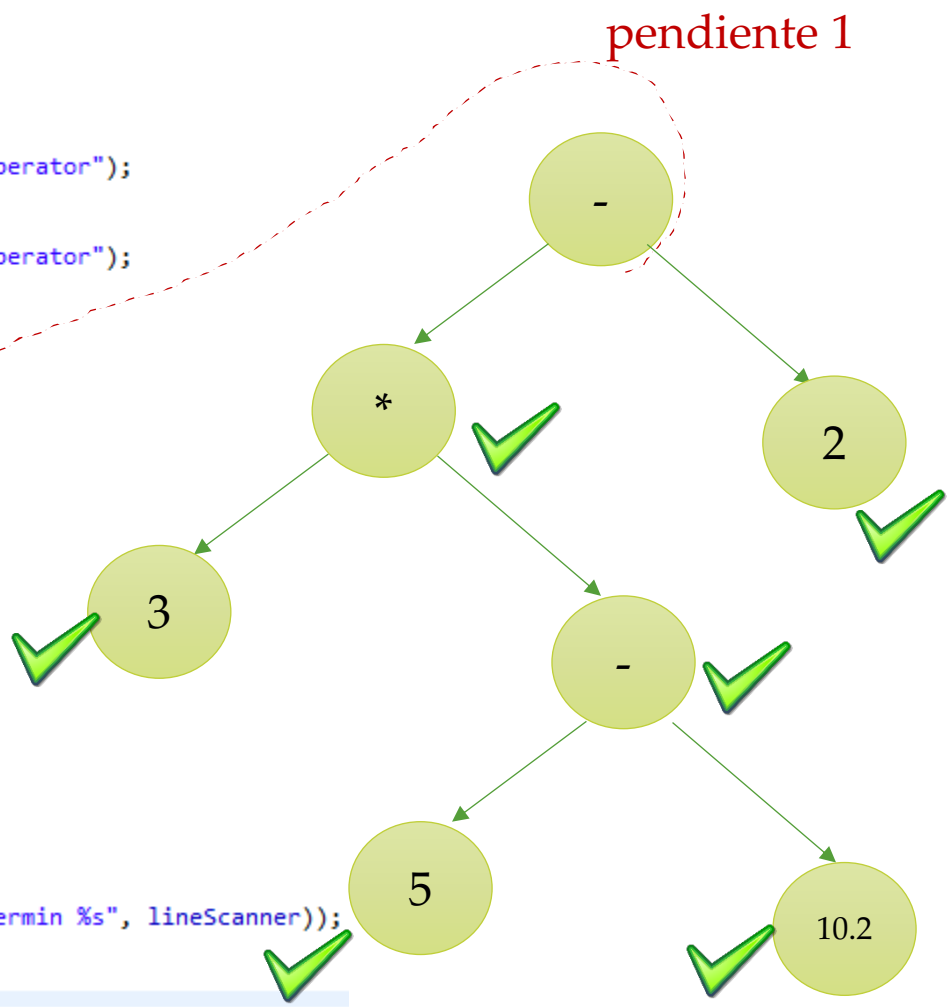
)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



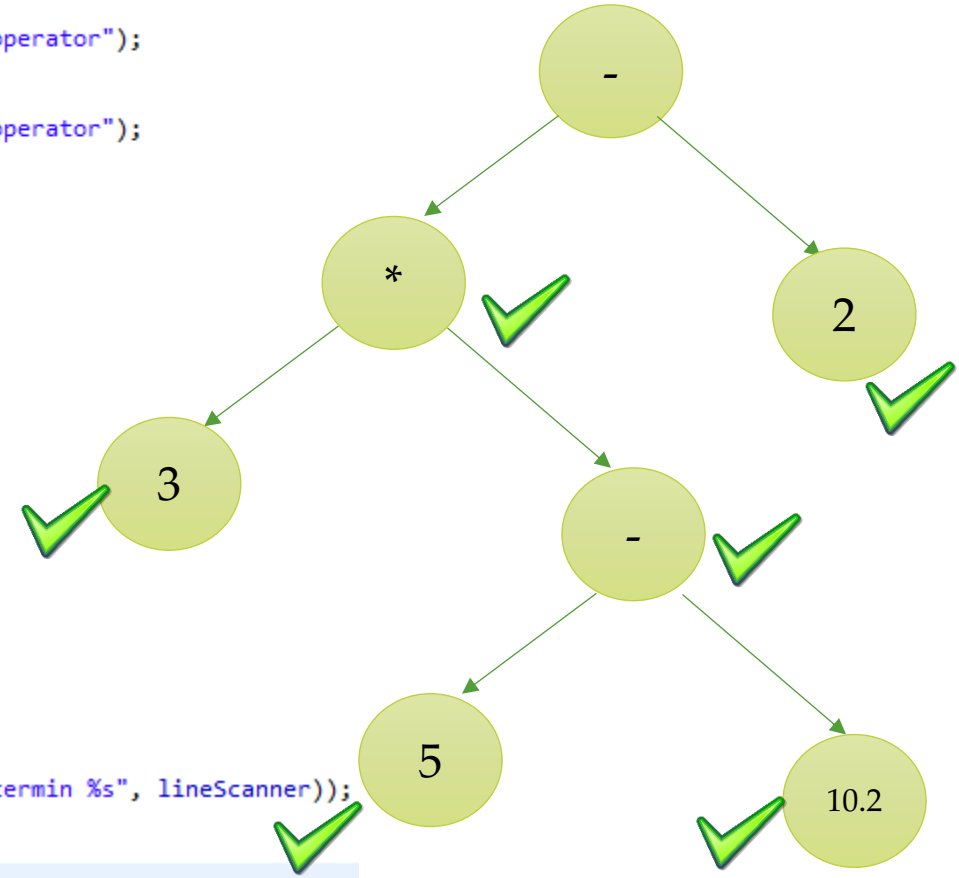
)

```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```

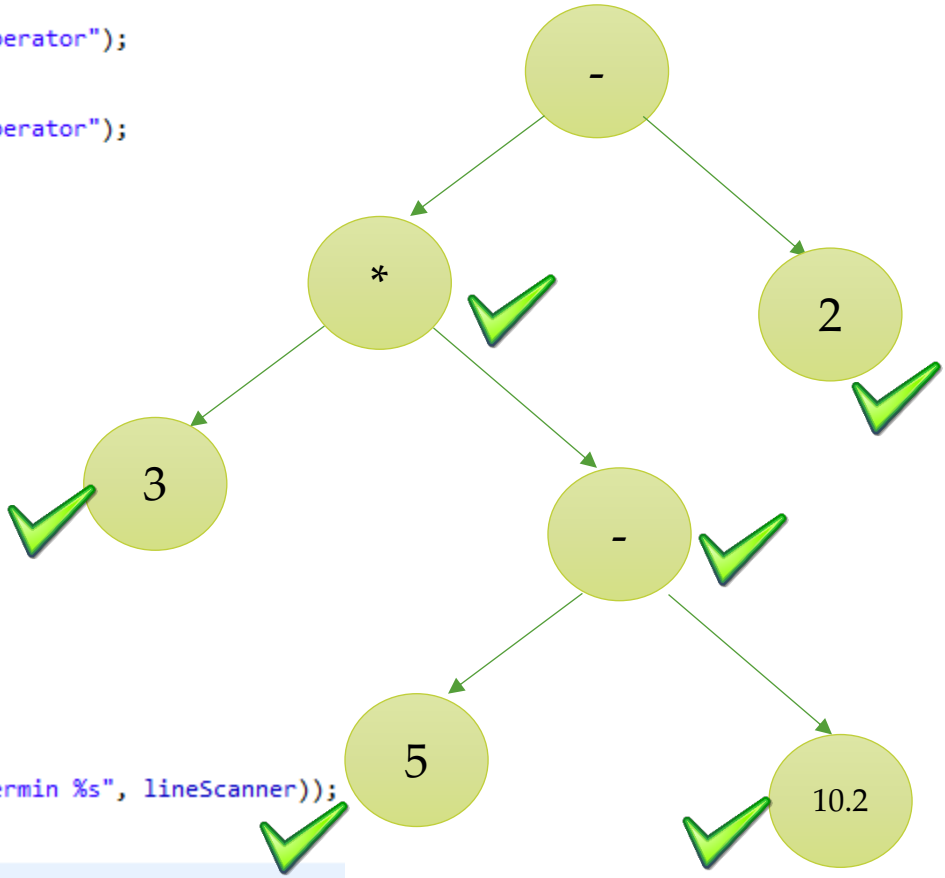


)

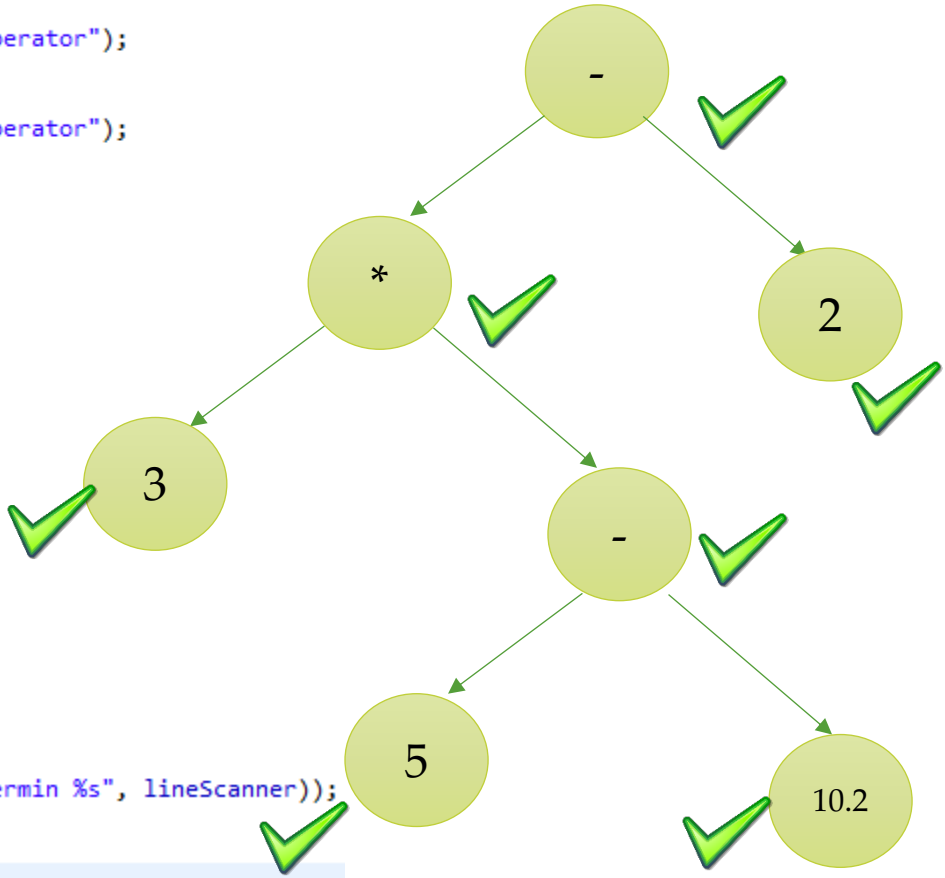
```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



```java
private Node buildExpression() {
        Node n = new Node();

        if (lineScanner.hasNext("\\(")) {
           lineScanner.next(); // lo consumo

           n.left = buildExpression(); // subexpression

           // operator
           if (!lineScanner.hasNext())
              throw new RuntimeException("missing or invalid operator");
           n.data = lineScanner.next();
           if (!Utils.isOperator(n.data))
              throw new RuntimeException("missing or invalid operator");

           // subexpression
           n.right = buildExpression();

           // ) expected
           if (lineScanner.hasNext("\\)")) {
              // lo consumo
              lineScanner.next();
           } else {
              throw new RuntimeException("missing )" );
           }

           return n;
        }

        // constant
        if (!lineScanner.hasNext())
           throw new RuntimeException("missing expression");

        n.data = lineScanner.next();
        if (!Utils.isConstant(n.data)) {
           throw new RuntimeException(String.format("illegal termin %s", lineScanner));
        }
        return n;
    }
```



Seguimiento del método recursivo buildExpression() para "((3 * (5 - 10.2)) - 2)"

E -> (E op E)
E -> cte