IZRAZNA DREVESA: IMPLEMENTACIJA

```
public class ArithmeticExprNode {

   int operator ;
   double value ;
   ArithmeticExprNode left, right ;
} // class ArithmeticExprNode


public class ArithmeticExprTree {
   static final int PLUS = '+' ;
   static final int MINUS = '−' ;
   static final int TIMES = '∗' ;
   static final int DIVIDE = '/' ;
   static final int L_BRACKET = '(' ;
   static final int R_BRACKET = ')' ;
   static final int NUM_VALUE = '\0' ;

   ArithmeticExprNode rootNode ;

   } // class ArithmeticExprTree
```

Izrazna drevesa

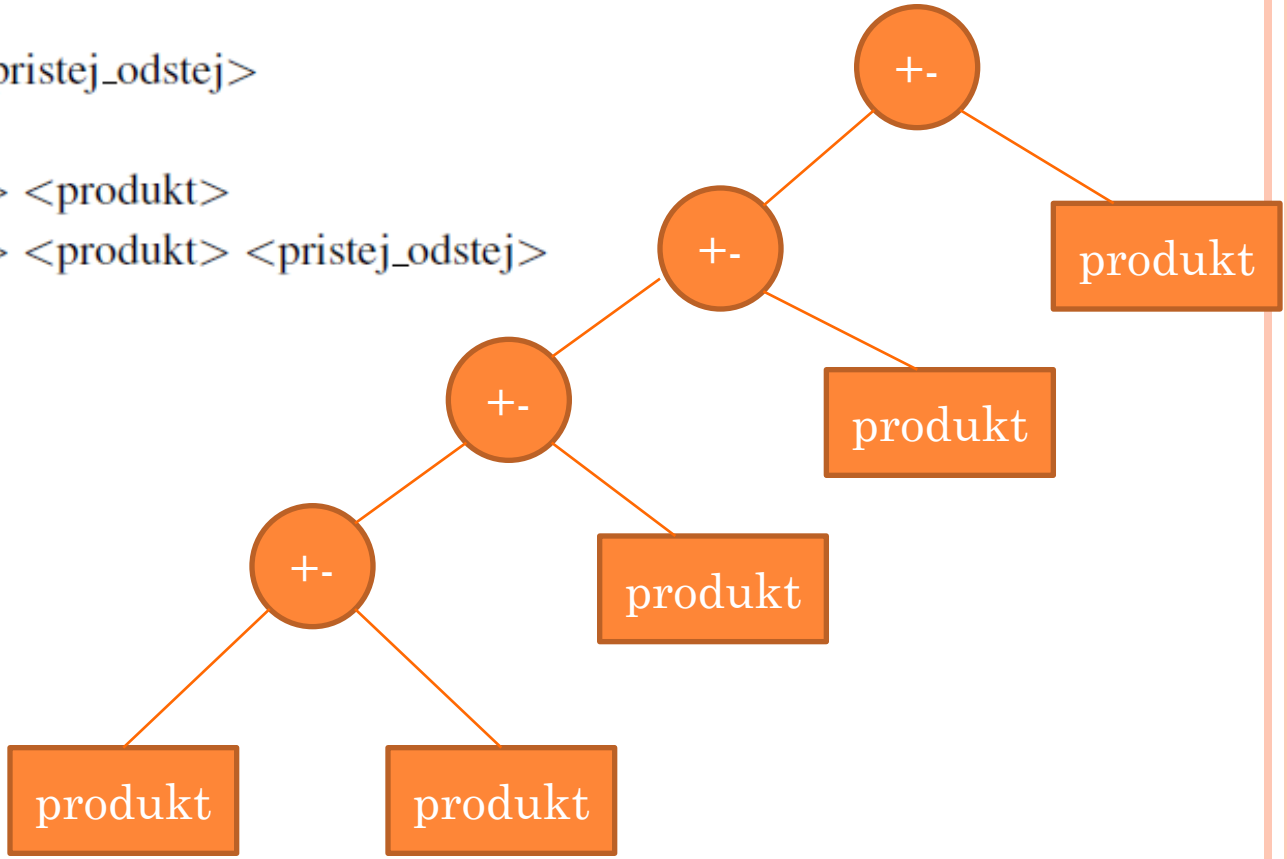
```
private double evaluate(ArithmeticExprNode node) {
  if (node == null)
    return 0.0 ; // alternatively throw exception ;
  else if (node.operator == NUM_VALUE)
    return node.value ;
    else {
      double lValue = evaluate(node.left) ;
      double rValue = evaluate(node.right) ;
      switch (node.operator) {
        case PLUS: return lValue + rValue ;
        case MINUS: return lValue − rValue ;
        case TIMES: return lValue ∗ rValue ;
        case DIVIDE: return lValue / rValue ;
        default: return 0.0 ; // alternatively throw exception ;
      } // switch
    } // else
  } // evaluate
```

KONTEKSTNO NEODVISNA GRAMATIKA

<Izraz> ::= <produkt>
<izraz> ::= <produkt> <pristej_odstej>

<pristej_odstej> ::= <+−> <produkt>
<pristej_odstej> ::= <+−> <produkt> <pristej_odstej>

<+−> ::= +
<+−> ::= −



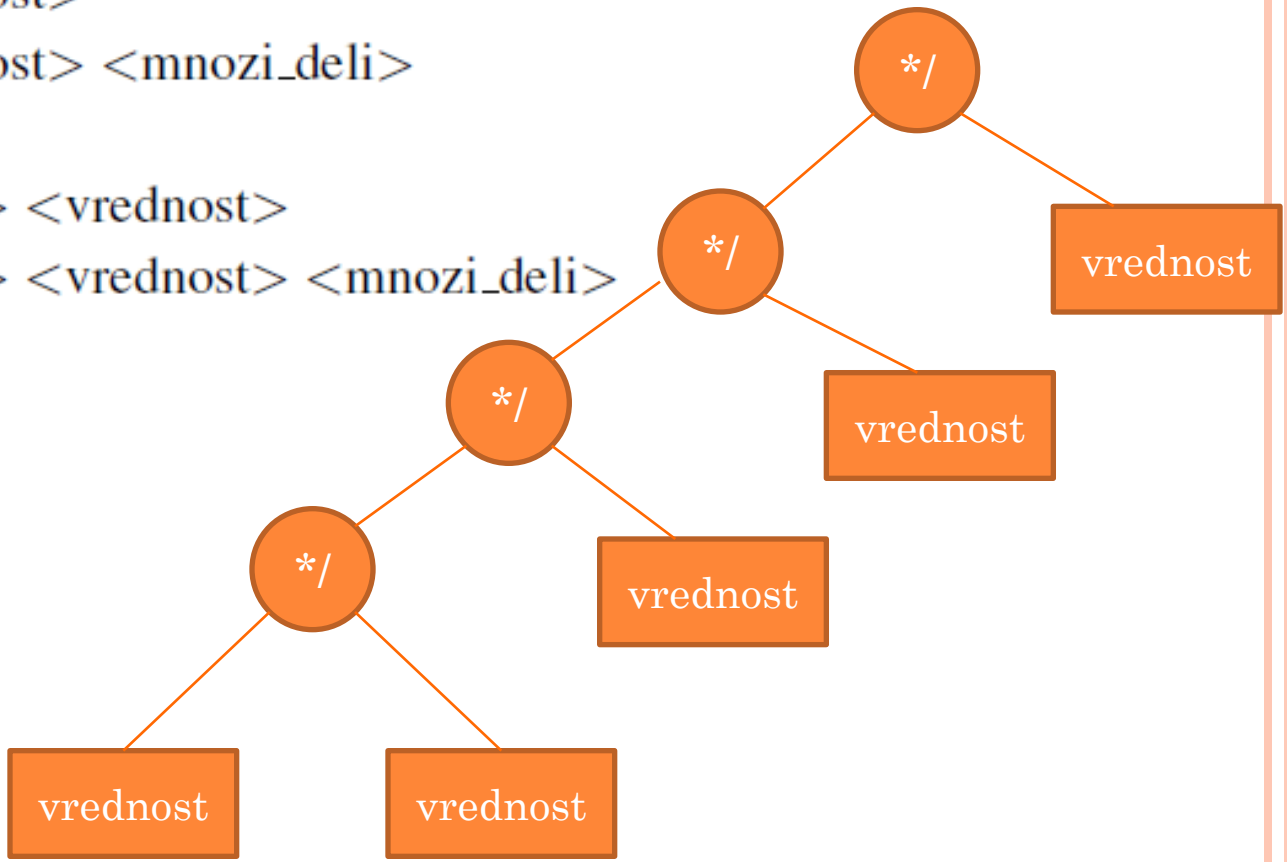
<izraz> → <p><po> → <p><+-><p><po> → <p><+-><p><+-><p><po> →
<p><+-><p><+-><p><+-><p><po> → <p><+-><p><+-><p><+-><p><+-><p>

KONTEKSTNO NEODVISNA GRAMATIKA

<produkt> ::= <vrednost>
<produkt> ::= <vrednost> <mnozi_deli>

<mnozi_deli> ::= <*/> <vrednost>
<mnozi_deli> ::= <*/> <vrednost> <mnozi_deli>

<*/> ::= *
<*/> ::= /



KONTEKSTNO NEODVISNA GRAMATIKA

Treba je definirati samo še <vrednost>

<vrednost> ::= <stevilo>

<vrednost> ::= (<izraz>)

KONTEKSTNO NEODVISNA GRAMATIKA

<Izraz> ::= <produkt>
<izraz> ::= <produkt> <pristej_odstej>

<pristej_odstej> ::= <+−> <produkt>
<pristej_odstej> ::= <+−> <produkt> <pristej_odstej>

<+−> ::= +
<+−> ::= −

<produkt> ::= <vrednost>
<produkt> ::= <vrednost> <mnozi_deli>

<mnozi_deli> ::= <∗/> <vrednost>
<mnozi_deli> ::= <∗/> <vrednost> <mnozi_deli>

<∗/> ::= ∗
<∗/> ::= /

<vrednost> ::= <stevilo>
<vrednost> ::= (<izraz>)

SINTAKTIČNA ANALIZA

Branje simbolov v javi: `StreamTokenizer`

`StreamTokenizer st; // st` je novi "bralec" simbolov

`st.nextToken(); //` prebere naslednji simbol

vrsta prebranega simbola `st.ttype` je lahko
- `PLUS`
- `MINUS`
- `TIMES`
- `DIVIDE`
- `L_BRACKET`
- `R_BRACKET`
- `TT_NUMBER`
- `TT_EOF //` konec vhodnega niza

Če je vrsta simbola `TT_NUMBER`,
je vrednost prebranega števila `st.nval`

IZRAZNA DREVESA: IMPLEMENTACIJA

```
public class ArithmeticExprNode {

   int operator ;
   double value ;
   ArithmeticExprNode left, right ;
} // class ArithmeticExprNode


public class ArithmeticExprTree {
   static final int PLUS = '+' ;
   static final int MINUS = '−' ;
   static final int TIMES = '∗' ;
   static final int DIVIDE = '/' ;
   static final int L_BRACKET = '(' ;
   static final int R_BRACKET = ')' ;
   static final int NUM_VALUE = '\0' ;

   ArithmeticExprNode rootNode ;

   } // class ArithmeticExprTree
```

Sintaktična analiza + drevo

Med sintaktično analizo gradimo izrazno drevo.

Vsaka metoda dobi:
- prvi (že prebrani) simbol
- eventuelne dele za gradnjo svojega (pod)drevesa

Vsaka metoda vrne
- naslednji prebrani simbol
- ustrezno poddrevo (kazalec na koren)
- eventuelne dele poddrevesa za naslednjo metodo

Glavna metoda prebere prvi simbol in pokliče metodo `expression`, na koncu pa še preveri zaključek analize:

```
public void exprToTree(StreamTokenizer st) throws IOException {
   st.nextToken() ;
   rootNode = expression(st) ;
   if (st.ttype != StreamTokenizer.TT_EOF)
      System.out.println("Expression error");
} // exprToTree
```

SINTAKTIČNA ANALIZA + DREVO

<Izraz> ::= <produkt>

<izraz> ::= <produkt> <pristej_odstej>

```
private ArithmeticExprNode expression(StreamTokenizer st)
                                                    throws IOException {
   ArithmeticExprNode root = product(st) ;
   if (st.ttype == PLUS || st.ttype == MINUS)
      return addSubtract(st, root) ;
   else return root ;
} // expression
```

SINTAKTIČNA ANALIZA + DREVO

<pristej_odstej> ::= <+−> <produkt>
<pristej_odstej> ::= <+−> <produkt> <pristej_odstej>

```
private ArithmeticExprNode addSubtract(StreamTokenizer st,
                                       ArithmeticExprNode left) throws IOException{

  ArithmeticExprNode root = new ArithmeticExprNode(st.ttype) ;
  root.left = left ;
  st.nextToken() ;
  root.right = product(st) ;
 if (st.ttype == PLUS || st.ttype == MINUS)
     return addSubtract(st, root) ;
 else return root ;
} // addSubtract
```

Gramatika določa program

<Izraz> ::= <produkt>

<izraz> ::= <produkt> <pristej_odstej>

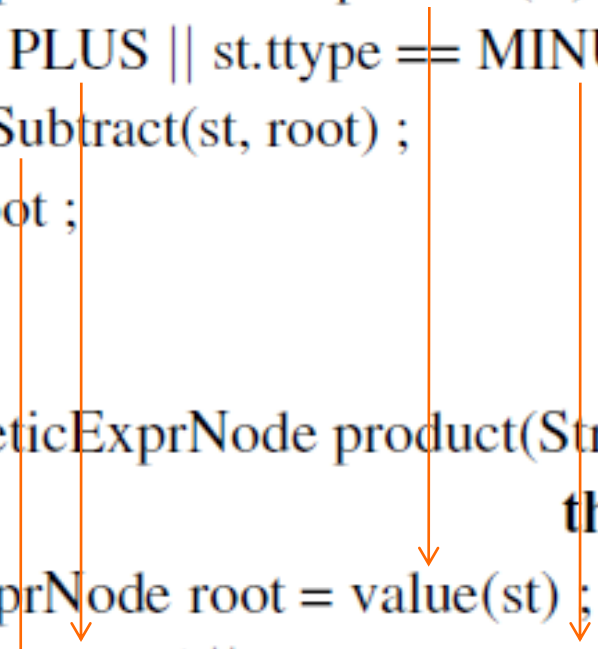
<produkt> ::= <vrednost>

<produkt> ::= <vrednost> <mnozi_deli>

Gramatika določa program

```
private ArithmeticExprNode expression(StreamTokenizer st)
                                                    throws IOException {
   ArithmeticExprNode root = product(st) ;
   if (st.ttype == PLUS || st.ttype == MINUS)
      return addSubtract(st, root) ;
   else return root ;
} // expression

private ArithmeticExprNode product(StreamTokenizer st)
                                                    throws IOException {
   ArithmeticExprNode root = value(st) ;
   if (st.ttype == TIMES || st.ttype ==DIVIDE)
       return multiplyDivide(st, root) ;
   else return root ;
} // product
```



Gramatika določa program

<pristej_odstej> ::= <+−> <produkt>
<pristej_odstej> ::= <+−> <produkt> <pristej_odstej>

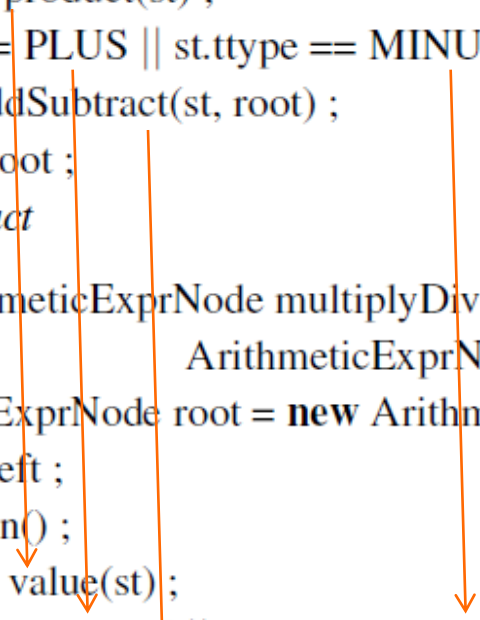
<mnozi_deli> ::= <∗/> <vrednost>
<mnozi_deli> ::= <∗/> <vrednost> <mnozi_deli>

GRAMATIKA DOLOČA PROGRAM

```
private ArithmeticExprNode addSubtract(StreamTokenizer st,
                                       ArithmeticExprNode left) throws IOException{

   ArithmeticExprNode root = new ArithmeticExprNode(st.ttype) ;
   root.left = left ;
   st.nextToken() ;
   root.right = product(st) ;
 if (st.ttype == PLUS || st.ttype == MINUS)
     return addSubtract(st, root) ;
 else return root ;
} // addSubtract

private ArithmeticExprNode multiplyDivide(StreamTokenizer st,
                                          ArithmeticExprNode left) throws IOException{
   ArithmeticExprNode root = new ArithmeticExprNode(st.ttype) ;
   root.left = left ;
   st.nextToken() ;
   root.right = value(st) ;
   if (st.ttype == TIMES || st.ttype == DIVIDE) // product continues
     return multiplyDivide(st, root) ;
   else return root ;
} // multiplyDivide
```



SINTAKTIČNA ANALIZA + DREVO

```
private ArithmeticExprNode value(StreamTokenizer st) throws IOException {
    if (st.ttype == StreamTokenizer.TT_NUMBER) {
       ArithmeticExprNode node =
                                  new ArithmeticExprNode(NUM_VALUE, st.nval);
       st.nextToken() ;
       return node ;
    }
    else if (st.ttype != L_BRACKET) // should be left bracket
       return null ; // error, alternatively throw exception
    else {
       st.nextToken() ;
       ArithmeticExprNode node = expression(st) ; // indirect recursion
       if (st.ttype != R_BRACKET) return null ;
       else {
          st.nextToken();
          return node ;
       }
    }
} // value
```

<vrednost> ::= <stevilo>

<vrednost> ::= (<izraz>)

REKURZIJA → ITERACIJA

```
private ArithmeticExprNode addSubtract(StreamTokenizer st,
                                       ArithmeticExprNode left) throws IOException{

   ArithmeticExprNode root = new ArithmeticExprNode(st.ttype) ;
   root.left = left ;
   st.nextToken() ;
   root.right = product(st) ;
 if (st.ttype == PLUS || st.ttype == MINUS)
      return addSubtract(st, root) ;
 else return root ;
} // addSubtract

private ArithmeticExprNode multiplyDivide(StreamTokenizer st,
                                          ArithmeticExprNode left) throws IOException{
   ArithmeticExprNode root = new ArithmeticExprNode(st.ttype) ;
   root.left = left ;
   st.nextToken() ;
   root.right = value(st) ;
   if (st.ttype == TIMES || st.ttype == DIVIDE) // product continues
      return multiplyDivide(st, root) ;
   else return root ;
} // multiplyDivide
```

Rekurzija → iteracija

<Izraz> ::= <produkt>
<izraz> ::= <produkt> <pristej_odstej>

<pristej_odstej> ::= <+−> <produkt>
<pristej_odstej> ::= <+−> <produkt> <pristej_odstej>

<izraz> ::= <produkt> { <+−> <produkt> }

IZZIV

<izraz> ::= <produkt> { <+−> <produkt> }

<+−> ::= +

<+−> ::= −

<produkt> ::= <vrednost> { <∗/> <vrednost> }

<∗/> ::= ∗

<∗/> ::= /

<vrednost> ::= <stevilo>

<vrednost> ::= (<izraz>)