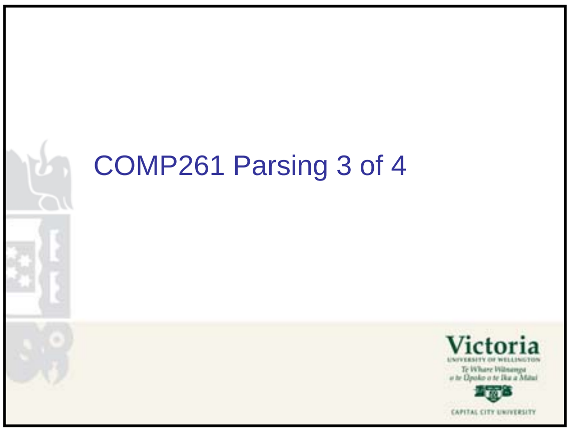
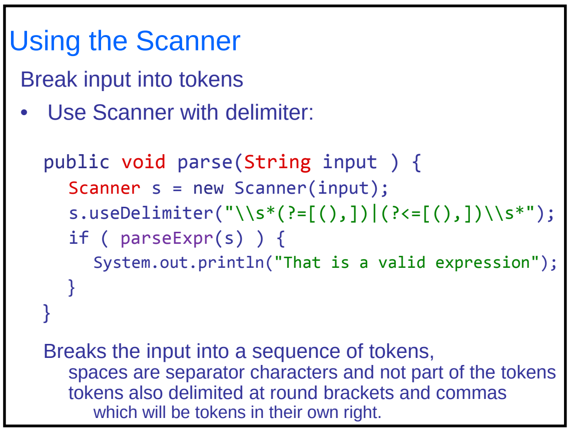


COMP261 Parsing 3 of 4

Victoria
University of Wellington
Te Whare Wānanga o te Ūpoko o te Ika a Māui
Capital City University



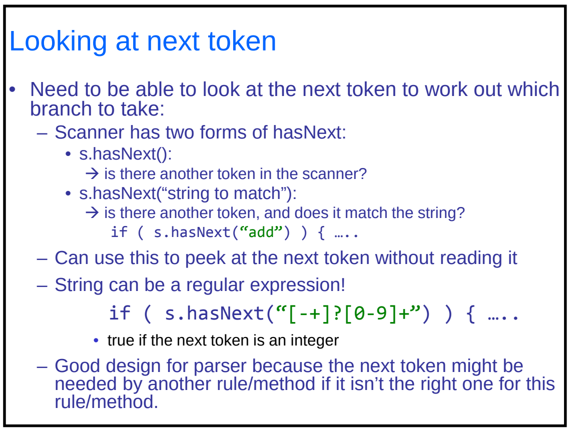
Using the Scanner

Break input into tokens

- Use Scanner with delimiter:

```
public void parse(String input ) {
   Scanner s = new Scanner(input);
   s.useDelimiter("\\s*(?=[(),])|(?<=[(),])\\s*");
   if ( parseExpr(s) ) {
      System.out.println("That is a valid expression");
   }
}
```

Breaks the input into a sequence of tokens,
spaces are separator characters and not part of the tokens
tokens also delimited at round brackets and commas
which will be tokens in their own right.

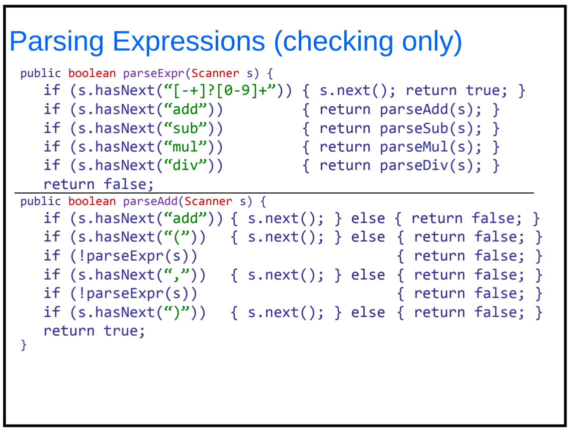


Looking at next token

- Need to be able to look at the next token to work out which branch to take:
  - Scanner has two forms of hasNext:
    - s.hasNext():
      → is there another token in the scanner?
    - s.hasNext("string to match"):
      → is there another token, and does it match the string?
      `if ( s.hasNext("add") ) { …..`
  - Can use this to peek at the next token without reading it
  - String can be a regular expression!

    ```
    if ( s.hasNext("[-+]?[0-9]+") ) { …..
    ```

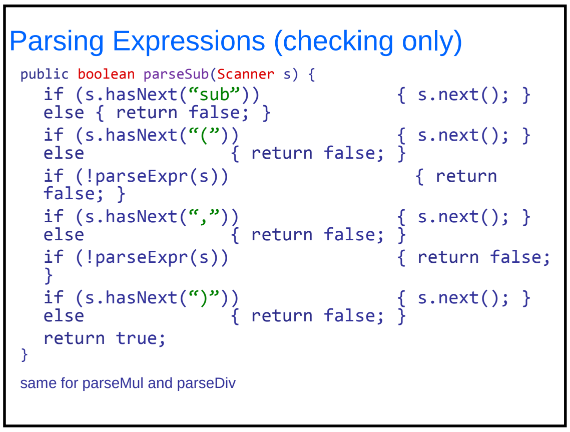
    - true if the next token is an integer
  - Good design for parser because the next token might be needed by another rule/method if it isn't the right one for this rule/method.



Parsing Expressions (checking only)

```
public boolean parseExpr(Scanner s) {
   if (s.hasNext("[-+]?[0-9]+")) { s.next(); return true; }
   if (s.hasNext("add"))         { return parseAdd(s); }
   if (s.hasNext("sub"))         { return parseSub(s); }
   if (s.hasNext("mul"))         { return parseMul(s); }
   if (s.hasNext("div"))         { return parseDiv(s); }
   return false;
```

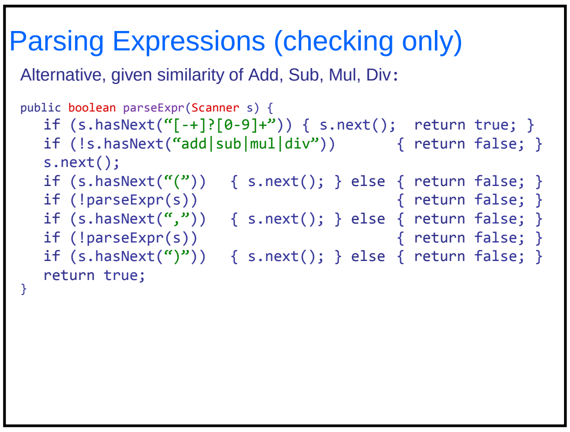
```
public boolean parseAdd(Scanner s) {
   if (s.hasNext("add")) { s.next(); } else { return false; }
   if (s.hasNext("("))   { s.next(); } else { return false; }
   if (!parseExpr(s))                        { return false; }
   if (s.hasNext(","))   { s.next(); } else { return false; }
   if (!parseExpr(s))                        { return false; }
   if (s.hasNext(")"))   { s.next(); } else { return false; }
   return true;
}
```



Parsing Expressions (checking only)

```
public boolean parseSub(Scanner s) {
   if (s.hasNext("sub"))              { s.next(); }
   else { return false; }
   if (s.hasNext("("))                { s.next(); }
   else                    { return false; }
   if (!parseExpr(s))                   { return
   false; }
   if (s.hasNext(","))                { s.next(); }
   else                    { return false; }
   if (!parseExpr(s))                 { return false;
   }
   if (s.hasNext(")"))                { s.next(); }
   else                    { return false; }
   return true;
}
```

same for parseMul and parseDiv



Parsing Expressions (checking only)

Alternative, given similarity of Add, Sub, Mul, Div:

```
public boolean parseExpr(Scanner s) {
   if (s.hasNext("[-+]?[0-9]+")) { s.next();  return true; }
   if (!s.hasNext("add|sub|mul|div"))        { return false; }
   s.next();
   if (s.hasNext("("))   { s.next(); } else { return false; }
   if (!parseExpr(s))                        { return false; }
   if (s.hasNext(","))   { s.next(); } else { return false; }
   if (!parseExpr(s))                        { return false; }
   if (s.hasNext(")"))   { s.next(); } else { return false; }
   return true;
}
```

How do we construct a parse tree?

- Given our grammar:

  Expr ::= Num | Add | Sub | Mul | Div
  Add ::= “add” “(” Expr “,” Expr “)”
  Sub ::= “sub” “(“ Expr “,” Expr “)”
  Mul ::= “mul” “(” Expr “,” Expr “)”
  Div ::= “div” “(“ Expr “,” Expr “)”
  Num ::= an optional sign followed by a sequence of digits:
  [-+]?[0-9]+

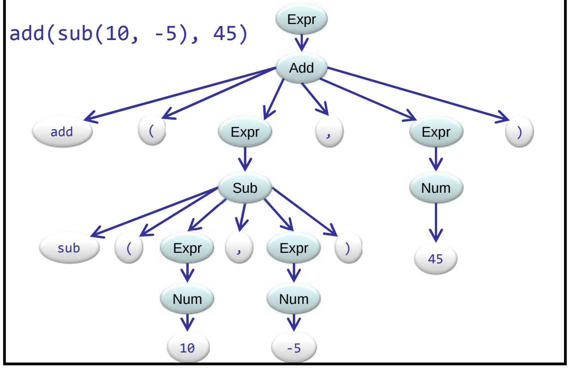
- And an expression:

  `add(sub(10, -5), 45)`

- First goal is a concrete parse tree:

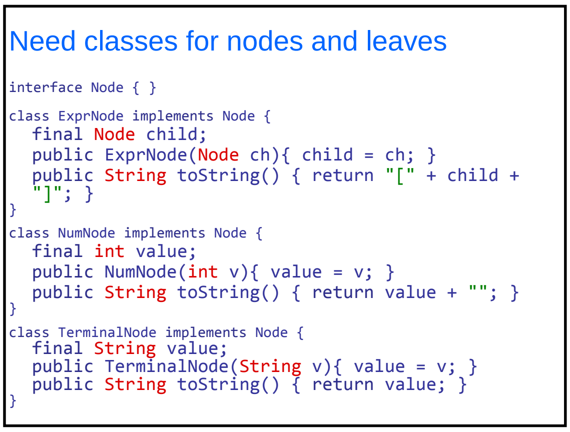
How do we construct a parse tree?

`add(sub(10, -5), 45)`



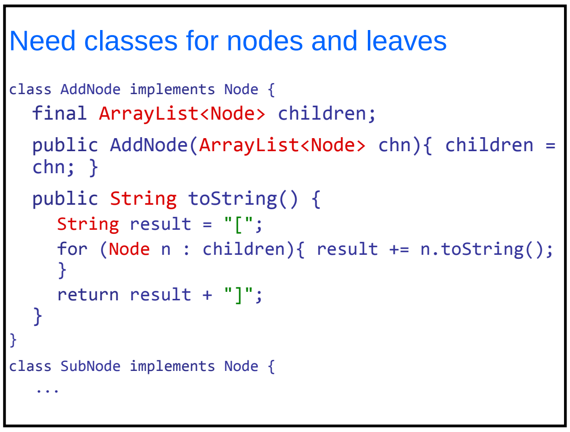
Modifying parser to produce parse tree

- Need to have Node classes to represent the syntax tree
  - Expression Nodes
    - contain a number or an Add/Sub/Mul/Div
  - Add, Sub, Mul, Div nodes
  - Number Nodes
  - Terminal Nodes
    - for the terminal values
    - just contain a string.



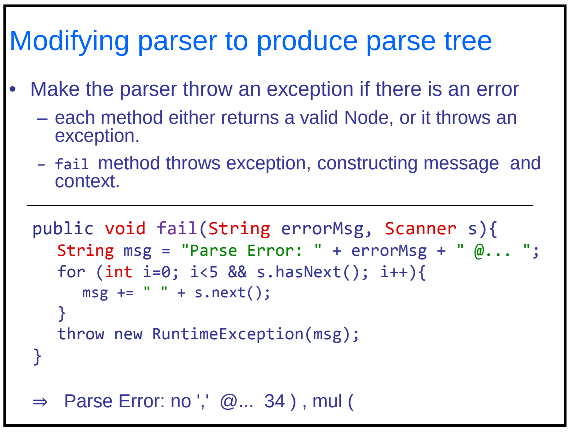
Need classes for nodes and leaves

```
interface Node { }
class ExprNode implements Node {
  final Node child;
  public ExprNode(Node ch){ child = ch; }
  public String toString() { return "[" + child +
  "]"; }
}
class NumNode implements Node {
  final int value;
  public NumNode(int v){ value = v; }
  public String toString() { return value + ""; }
}
class TerminalNode implements Node {
  final String value;
  public TerminalNode(String v){ value = v; }
  public String toString() { return value; }
}
```



Need classes for nodes and leaves

```
class AddNode implements Node {
  final ArrayList<Node> children;
  public AddNode(ArrayList<Node> chn){ children =
  chn; }
  public String toString() {
    String result = "[";
    for (Node n : children){ result += n.toString();
    }
    return result + "]";
  }
}
class SubNode implements Node {
  ...
```



Modifying parser to produce parse tree

- Make the parser throw an exception if there is an error
  - each method either returns a valid Node, or it throws an exception.
  - `fail` method throws exception, constructing message and context.

```
public void fail(String errorMsg, Scanner s){
  String msg = "Parse Error: " + errorMsg + " @... ";
  for (int i=0; i<5 && s.hasNext(); i++){
    msg += " " + s.next();
  }
  throw new RuntimeException(msg);
}
```

⇒ Parse Error: no ',' @... 34) , mul (

Modifying parser to produce parse tree

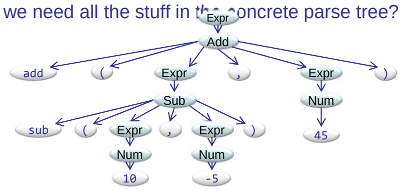
```
public Node parseExpr(Scanner s) {
   if (!s.hasNext())           { fail("Empty expr",s); }
   Node child = null;
   if (s.hasNext("-?\\d+"))    { child = parseNumNode(s);}
   else if (s.hasNext("add"))  { child = parseAddNode(s); }
   else if (s.hasNext("sub"))  { child = parseSubNode(s); }
   else if (s.hasNext("mul"))  { child = parseMulNode(s); }
   else if (s.hasNext("div"))  { child = parseDivNode(s); }
   else { fail("not an expression", s); }
   return new ExprNode(child);
}

public Node parseNumNode(Scanner s) {
   if (!s.hasNextInt())        { fail("not an integer", s); }
   return new NumNode(s.nextInt());
}
```

Modifying parser to produce parse tree

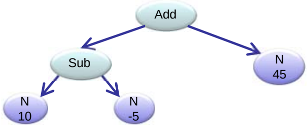
```
public Node parseAddNode(Scanner s) {
   ArrayList<Node> children = new ArrayList<Node>();
   if (!s.hasNext("add"))        { fail("no ‘add’", s);
   }
   children.add(new TerminalNode(s.next()));
   if (!s.hasNext("("))          { fail("no ‘(’", s); }
   children.add(new TerminalNode(s.next()));
   children.add(parseExpr(s));
   if (!s.hasNext(","))          { fail("no ‘,’", s); }
   children.add(new TerminalNode(s.next()));
   children.add(parseExpr(s));
   if (!s.hasNext(")"))          { fail("no ‘)’", s); }
   children.add(new TerminalNode(s.next()));
   return new ExprNode(children);
}
```

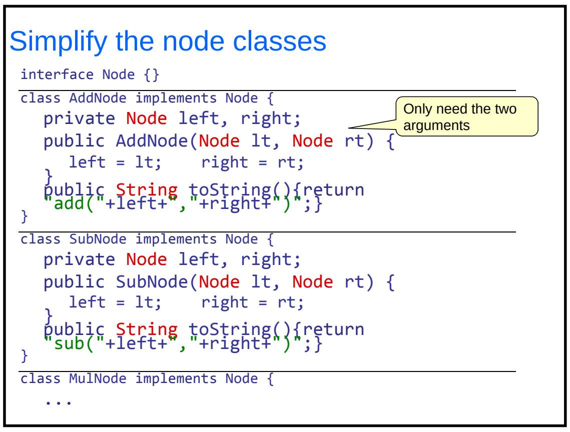
What about abstract syntax trees?

- Do we need all the stuff in the concrete parse tree?



- An abstract syntax tree:
- Don't need
  - literal strings from rules
  - useless nodes
    - Expr
    - tokens under Num

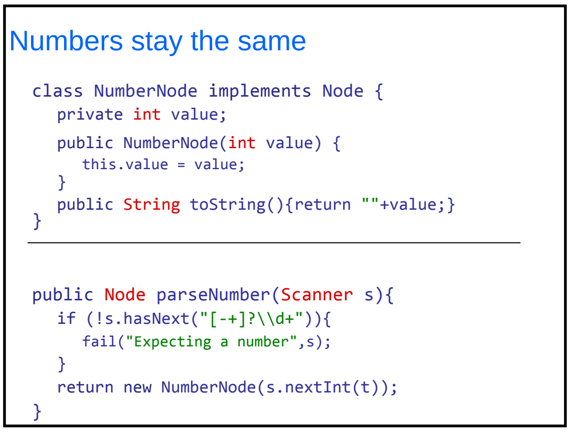




Simplify the node classes

```
interface Node {}
class AddNode implements Node {
   private Node left, right;
   public AddNode(Node lt, Node rt) {
      left = lt;    right = rt;
   }
   public String toString(){return
   "add("+left+","+right+")";}
}
class SubNode implements Node {
   private Node left, right;
   public SubNode(Node lt, Node rt) {
      left = lt;    right = rt;
   }
   public String toString(){return
   "sub("+left+","+right+")";}
}
class MulNode implements Node {
   ...
```

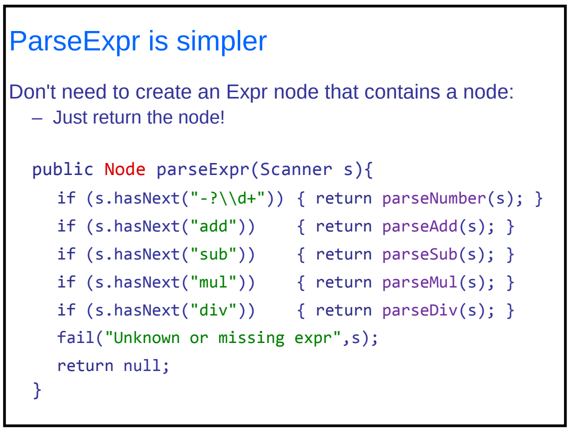
Only need the two arguments



Numbers stay the same

```
class NumberNode implements Node {
   private int value;
   public NumberNode(int value) {
      this.value = value;
   }
   public String toString(){return ""+value;}
}
```

```
public Node parseNumber(Scanner s){
   if (!s.hasNext("[-+]?\\d+")){
      fail("Expecting a number",s);
   }
   return new NumberNode(s.nextInt(t));
}
```

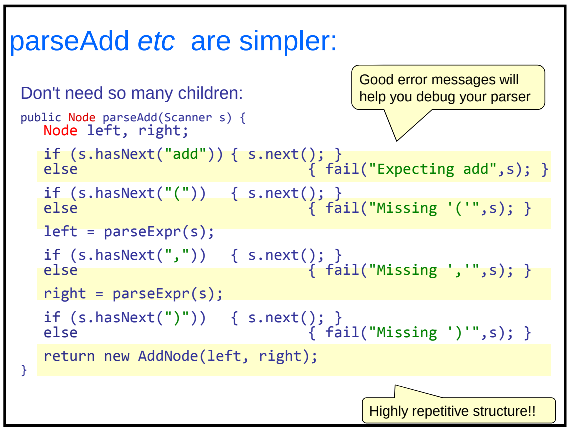


ParseExpr is simpler

Don't need to create an Expr node that contains a node:

- Just return the node!

```
public Node parseExpr(Scanner s){
   if (s.hasNext("-?\\d+")) { return parseNumber(s); }
   if (s.hasNext("add"))    { return parseAdd(s); }
   if (s.hasNext("sub"))    { return parseSub(s); }
   if (s.hasNext("mul"))    { return parseMul(s); }
   if (s.hasNext("div"))    { return parseDiv(s); }
   fail("Unknown or missing expr",s);
   return null;
}
```



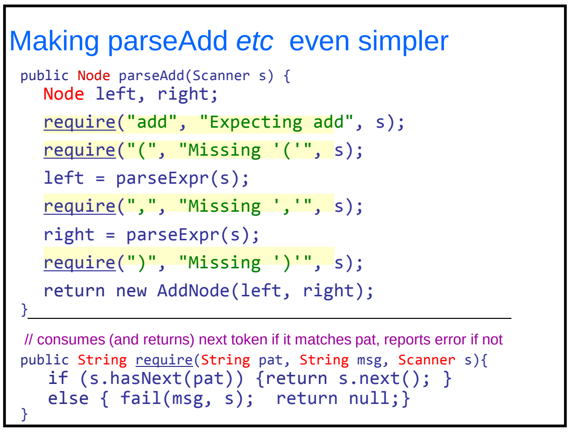
parseAdd *etc* are simpler:

Don't need so many children:

Good error messages will help you debug your parser

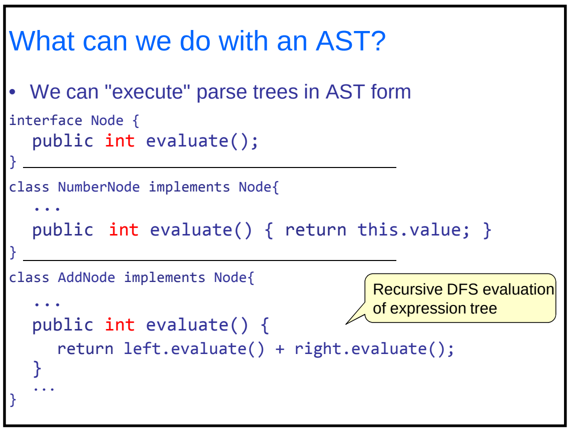
```
public Node parseAdd(Scanner s) {
   Node left, right;
   if (s.hasNext("add")) { s.next(); }
   else                           { fail("Expecting add",s); }
   if (s.hasNext("("))   { s.next(); }
   else                           { fail("Missing '('",s); }
   left = parseExpr(s);
   if (s.hasNext(","))   { s.next(); }
   else                           { fail("Missing ','",s); }
   right = parseExpr(s);
   if (s.hasNext(")"))   { s.next(); }
   else                           { fail("Missing ')'",s); }
   return new AddNode(left, right);
}
```

Highly repetitive structure!!



Making parseAdd *etc* even simpler

```
public Node parseAdd(Scanner s) {
   Node left, right;
   require("add", "Expecting add", s);
   require("(", "Missing '('", s);
   left = parseExpr(s);
   require(",", "Missing ','", s);
   right = parseExpr(s);
   require(")", "Missing ')'", s);
   return new AddNode(left, right);
}

// consumes (and returns) next token if it matches pat, reports error if not
public String require(String pat, String msg, Scanner s){
   if (s.hasNext(pat)) {return s.next(); }
   else { fail(msg, s);  return null;}
}
```



What can we do with an AST?

- We can "execute" parse trees in AST form

```
interface Node {
   public int evaluate();
}
class NumberNode implements Node{
   ...
   public  int evaluate() { return this.value; }
}
class AddNode implements Node{
   ...
   public int evaluate() {
      return left.evaluate() + right.evaluate();
   }
   ...
}
```

Recursive DFS evaluation of expression tree

What can we do with AST?

- We can print expressions in other forms

```
class AddNode implements Node {
  private Node left, right;
  public AddNode(Node lt, Node rt) {
    left = lt;
    right = rt;
  }
  public int evaluate() {
    return left.evaluate() + right.evaluate();
  }
  public String toString(){
    return "(" + left + " + " +
  }
}
```

Prints in regular infix notation (with brackets)

Nicer Language

- Allow floating point numbers as well as integers
  - need more complex pattern for numbers.

```
  class NumberNode implements Node {
    final double value;
    public NumberNode(double v){
      value= v;
    }
    public String toString(){
      return String.format("%.5f", value);
    }
    public double evaluate(){ return value; }
  }
```

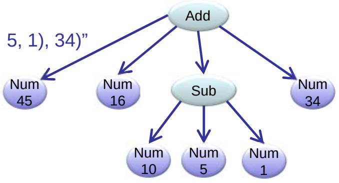
Nicer Language

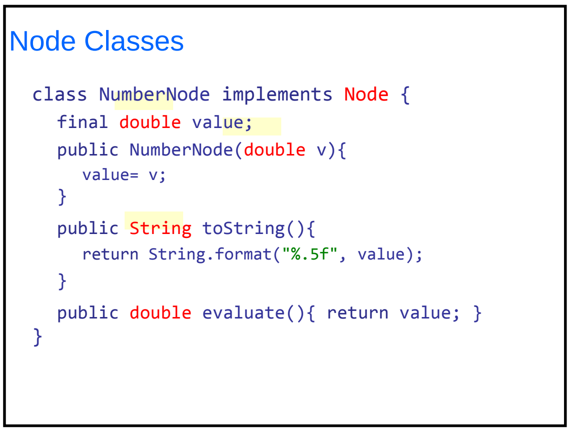
- Extend the language to allow 2 or more arguments:

```
Expr ::=   Num   | Add  |  Sub  | Mul  |  Div
Add  ::=  "add"  "("  Expr  [ ","  Expr ]+  ")"
Sub  ::= "sub"  "("  Expr  [ ","  Expr ]+  ")"
Mul   ::= "mul"  "("  Expr  [ ","  Expr ]+  ")"
Div    ::= "div"   "("  Expr  [ ","  Expr ]+  ")"
```

sub(16, 8, 2, 1) = 16 – 8 – 2 – 1

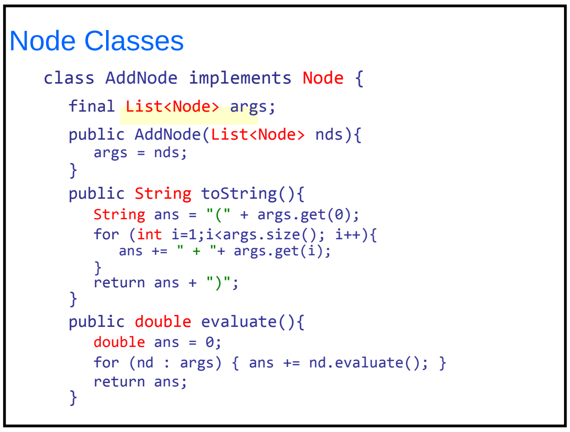
"add(45, 16, sub(10, 5, 1), 34)"





Node Classes

```
class NumberNode implements Node {
   final double value;
   public NumberNode(double v){
      value= v;
   }
   public String toString(){
      return String.format("%.5f", value);
   }
   public double evaluate(){ return value; }
}
```



Node Classes

```
 class AddNode implements Node {
    final List<Node> args;
    public AddNode(List<Node> nds){
       args = nds;
    }
    public String toString(){
       String ans = "(" + args.get(0);
       for (int i=1;i<args.size(); i++){
          ans += " + "+ args.get(i);
       }
       return ans + ")";
    }
    public double evaluate(){
       double ans = 0;
       for (nd : args) { ans += nd.evaluate(); }
       return ans;
    }
```