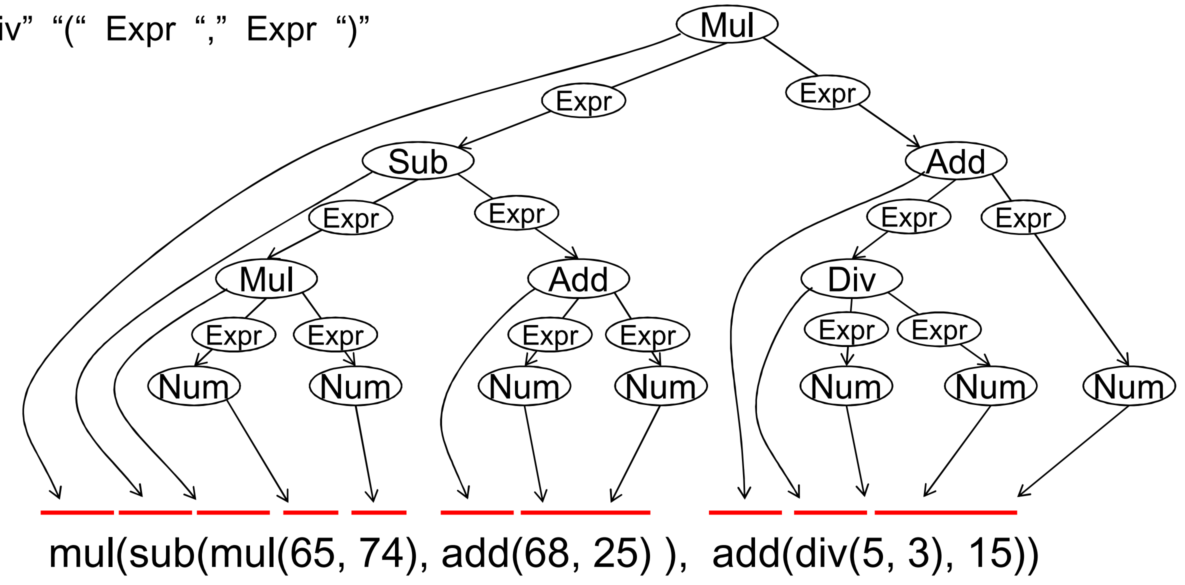
Admin

- Help desks start in week 3
- Tutorials start this week

A parser for arithmetic expressions

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



Using the Scanner

Break input into tokens

- Use Scanner with delimiter:

```
public void parse(String input ) {
   Scanner s = new Scanner(input);
   s.useDelimiter("\\s+|(?=[(),])|(?<=[(),])");
   if ( parseExpr(s) ) {System.out.println("That is a valid expression");}
   else                {System.out.println("That expression is NOT valid");}
}
```

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)

Expr ::= Num | Add | Sub | Mul | Div

```
public boolean parseExpr(Scanner s) {
   if (s.hasNext("[-+]?[0-9]+")) { s.next(); return true; }     // Num
   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;
 }
```

Parsing Expressions (checking only)

Add ::= “add” “(” Expr “,” Expr “)”

```
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))     { }            else {return false; }
   if (s.hasNext(“,”))   {s.next();}    else {return false; }
   if (parseExpr(s))     { }            else {return false; }
   if (s.hasNext(“)”))   {s.next();}    else {return false; }
   return true;
}
```

Parsing Expressions (checking only)

Sub ::= “sub” “(“ Expr “,” Expr “)”

```
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))     { }         else {return false;}
   if (s.hasNext(“,”))   {s.next();} else {return false;}
   if (parseExpr(s))     { }         else {return false;}
   if (s.hasNext(“)”))   {s.next();} else {return false;}
   return true;
}
```

same for parseMul and parseDiv

Cleaning up the code (checking only)

```
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))     { }            else {return false; }
   if (s.hasNext(","))   {s.next();}    else {return false; }
   if (parseExpr(s))     { }            else {return false; }
   if (s.hasNext(")"))   {s.next();}    else {return false; }
   return true;
}
```

```
// consumes next token if it matches pat, reports error if not
public boolean checkFor(String pat, Scanner s){
    if (s.hasNext(pat)) {s.next(); return true}
    else {return false;}
}
```

Cleaning up the code (checking only)

```
public boolean parseAdd(Scanner s) {
   if (!checkFor(“add”, s))   {return false; }
   if (!checkFor(“(”, s))     {return false; }
   if (!parseExpr(s))         {return false; }
   if (!checkFor(“,”, s))     {return false; }
   if (!parseExpr(s))         {return false; }
   if (!checkFor(“)”, s))     {return false; }
   return true;
}
```

```
// consumes next token if it matches pat, doesn't if not matching
public boolean checkFor(String pat, Scanner s){
    if (s.hasNext(pat)) {s.next(); return true}
    else {return false;}
}
```

Better coding: using patterns

- Give names to patterns to make program easier to understand and to modify
- Precompile the patterns for efficiency:

```
private static final Pattern NUM_PAT = Pattern.compile("[-+]?[0-9]+");
private static final Pattern ADD_PAT = Pattern.compile("add");
private static final Pattern SUB_PAT = Pattern.compile("sub");
private static final Pattern MUL_PAT = Pattern.compile("mul");
private static final Pattern DIV_PAT = Pattern.compile("div");
private static final Pattern OPEN_PAT = Pattern.compile("\\(");
private static final Pattern COMMA_PAT = Pattern.compile(",");
private static final Pattern CLOSE_PAT = Pattern.compile("\\)");
```

Using patterns (checking only)

```
public boolean parseAdd(Scanner s) {
   if (!checkFor(ADD_PAT, s))      {return false; }
   if (!checkFor(OPEN_PAT, s))     {return false; }
   if (!parseExpr(s))              {return false; }
   if (!checkFor(COMMA_PAT, s))    {return false; }
   if (!parseExpr(s))              {return false; }
   if (!checkFor(CLOSE_PAT, s))    {return false; }
   return true;
}
```

```
// consumes next token if it matches pat, doesn't if not matching
public boolean checkFor(Pattern pat, Scanner s){
    if (s.hasNext(pat)) {s.next(); return true}
    else {return false;}
}
```

Using Patterns (checking only)

Expr ::= Num | Add | Sub | Mul | Div

```
public boolean parseExpr(Scanner s) {
   if (s.hasNext(NUM_PAT))       { s.next(); return true; }     // Num
   if (s.hasNext(ADD_PAT))       { return parseAdd(s); }
   if (s.hasNext(SUB_PAT))       { return parseSub(s); }
   if (s.hasNext(MUL_PAT))       { return parseMul(s); }
   if (s.hasNext(DIV_PAT))       { return parseDiv(s); }
   return false;
 }
```

Generating an Abstract Syntax Tree

- Usually not enough to just say "Yes" or "No".
  - At least report what the error(s) were if it fails!
- Usually, need to construct the AST in order to use the text (program, webpage,…)
- Key ideas for recursive descent parser:
  - Parser should return a tree structure, ie, the root node of an AST
  - We need a node type for each Non-terminal (at least for the ones that matter)
  - Every parsing method should return a node for the tree.
  - The node from each recursive call should be added to the current node.

How do we construct a parse tree?

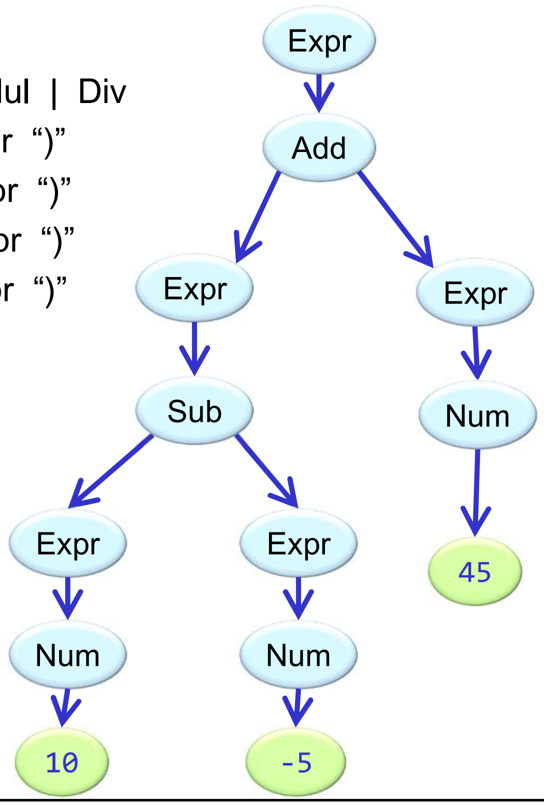
- Given our grammar:

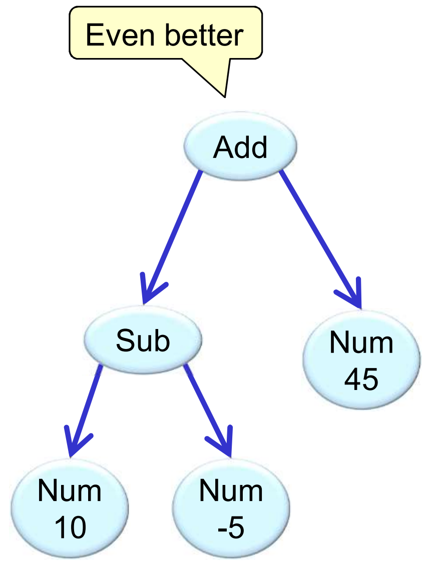
  Expr ::= Num | Add | Sub | Mul | Div
  Add ::= “add” “(” Expr “,” Expr “)”
  Sub ::= “sub” “(“ Expr “,” Expr “)”
  Mul ::= “mul” “(” Expr “,” Expr “)”
  Div ::= “div” “(“ Expr “,” Expr “)”
  Num ::= [-+]?[0-9]+

- And an expression:

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

- Build the tree





Modifying parser to produce parse tree

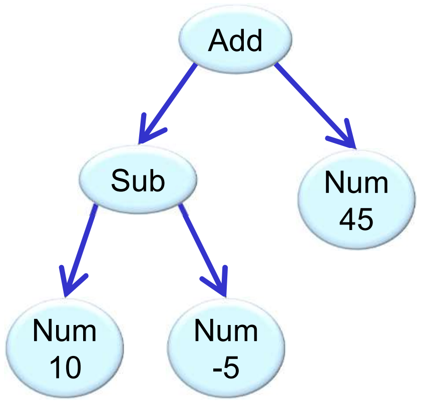
- Need an interface specifying the type of all the nodes
  - public interface ExprNode{…
- Need to have …Node classes to represent the types of each node.
  - Number Nodes
    - Contain the value.
  - Add, Sub, Mul, Div nodes
    - contain the ExprNodes for the two sub expressions
- Need to make the parse… methods return an ExprNode or throw exception
  - (instead of returning just true or false)

Need classes for nodes and leaves

```
public interface ExprNode { }

class NumNode implements ExprNode {
   final int value;
   public NumNode(int v){ value = v; }
}

class AddNode implements ExprNode {
   final ExprNode left;
   final ExprNode right;
   public AddNode(ExprNode lt, ExprNode rt){ left=lt; right=rt; }
}

class SubNode implements ExprNode {
   final ExprNode left;
   final ExprNode right;
   public SubNode(ExprNode lt, ExprNode rt){ left=lt; right=rt; }
}
```

Add ::= “add” “(” Expr “,” Expr “)”

Sub ::= “add” “(” Expr “,” Expr “)”



Modify the Parse… methods to return a parse tree or fail.

- Make the parser return an ExprNode if it is successful
- Make the parser throw an exception if there is an error
- `Use fail(…)` method to throw an exception with useful message.

```
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: Missing ',' @...34),mul(

parseExpr   (checking only)

Expr ::= Num | Add | Sub | Mul | Div

```
public boolean parseExpr(Scanner s) {
   if (!s.hasNext())             { return false; }
   if (s.hasNext(NUM_PAT)        { s.next(); return true; }
   if (s.hasNext(ADD_PAT))       { return parseAdd(s); }
   if (s.hasNext(SUB_PAT))       { return parseSub(s); }
   if (s.hasNext(MUL_PAT)        { return parseMul(s); }
   if (s.hasNext(DIV_PAT)        { return parseDiv(s); }
   return false;
 }
```

parseExpr returning a parse tree

Expr ::= Num | Add | Sub | Mul | Div

```
public ExprNode parseExpr(Scanner s) {
   if (!s.hasNext())              { fail("Empty expr",s); }
   if (s.hasNext(NUM_PAT))        { return parseNumNode(s);}
   if (s.hasNext(ADD_PAT))        { return parseAddNode(s); }
   if (s.hasNext(SUB_PAT))        { return parseSubNode(s); }
   if (s.hasNext(MUL_PAT))        { return parseMulNode(s); }
   if (s.hasNext(DIV_PAT))        { return parseDivNode(s); }
   fail("not an expression", s);
   return null;
}

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

parseAdd checking only

Add ::= “add” “(” Expr “,” Expr “)”

```
public boolean parseAdd(Scanner s) {
   if (!checkFor(ADD_PAT, s))     {return false; }
   if (!checkFor(OPEN_PAT, s))    {return false; }
   if (!parseExpr(s))             {return false; }
   if (!checkFor(COMMA_PAT, s))   {return false; }
   if (!parseExpr(s))             {return false; }
   if (!checkFor(CLOSE_PAT, s))   {return false; }
   return true;
}
```

parseAdd returning a node

Add ::= “add” “(” Expr “,” Expr “)”

```
public ExprNode parseAdd(Scanner s) {
   if (!checkFor(ADD_PAT, s))      {fail(“expecting ‘add’", s);}
   if (!checkFor(OPEN_PAT, s))     {fail(“missing ‘(’", s);}
   ExprNode left = parseExpr(s);
   if (!checkFor(COMMA_PAT, s))    {fail("missing ‘,’", s);}
   ExprNode right = parseExpr(s)
   if (!checkFor(CLOSE_PAT, s))    {fail("missing ‘)’", s);}
   return new AddNode(left, right);
}
```

Good error messages will help you debug your parser

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

parseAdd returning a node – simplified with require(…)

Add ::= “add” “(” Expr “,” Expr “)”

```
public ExprNode parseAdd(Scanner s) {
   require(ADD_PAT, “expecting ‘add’", s);
   require(OPEN_PAT, “missing ‘(’", s);
   ExprNode left = parseExpr(s);
   require(COMMA_PAT, "missing ‘,’", s);
   ExprNode right = parseExpr(s)
   require(CLOSE_PAT, "missing ‘)’", s);
   return new AddNode(left, right);
}
```

// consumes (and returns) next token if it matches pat, reports error if not

```
public String require(Pattern pat, String msg, Scanner s){
    if (s.hasNext(pat)) { return s.next(); }
    else { fail(msg, s);  return null;}
}
```