# COMP261 Parsing 3 of 4

Marcus Frean

## Coding a parser, in Java

Victoria
UNIVERSITY OF WELLINGTON
Te Whare Wānanga
o te Ūpoko o te Ika a Māui
CAPITAL CITY UNIVERSITY

# Today: exploring a recursive descent parser

- introducing our go-to example: a grammar for arithmetic expressions…

  eg: "add(-5, sub(50, 50), 4)"

- we will develop parser code for different goals:

1. just check a string for compliance
2. return a complete parse tree
3. return just the AST

This is intended to help you with Assignment #4.

# Tomorrow: when does this approach work?
# …or not?

# example: arithmetic expressions

- Consider this 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]+

- Check the following texts:

  div(100, 0)

  add(-5, sub(50, 50), 4)

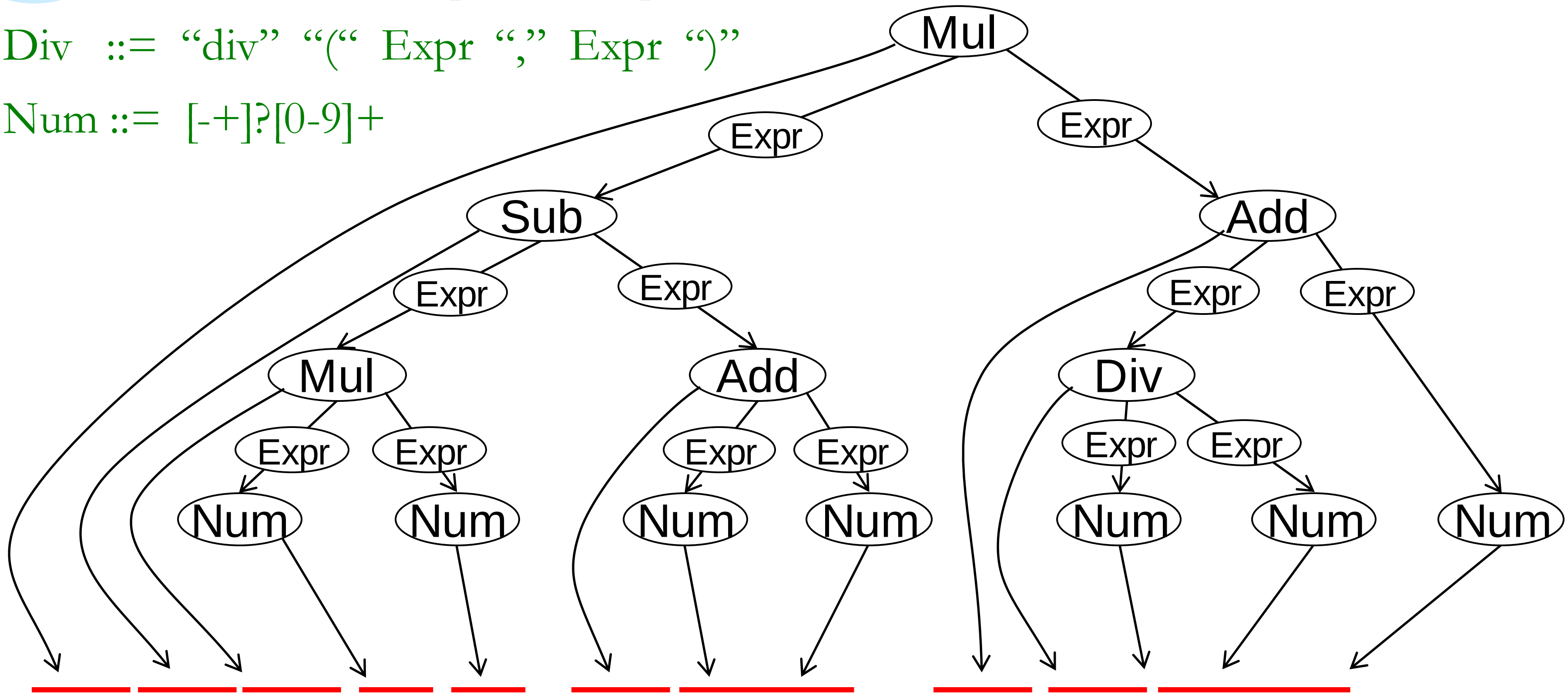
  div(div(86, 5), 67) 50

  mul(sub(mul(65, 74), add(68, 25) ), add(div(5, 3), 15))

let’s try this one

# an illustration

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]+




mul(sub(mul(65, 74), add(68, 25) ), add(div(5, 3), 15))

# Idea: Write a Program to Mimic Rules!

– have a method corresponding to each nonterminal that calls other nonterminal methods

For example, given a grammar:

FOO ::= “a” BAR | “b” BAZ

BAR ::= ….

Parser would have a method *something like* this:

```
public boolean parseFOO(Scanner s){
    if (!s.hasNext())              { return false; }          // PARSE ERROR
    String token = s.next();
    if (token.equals(“a”))         { return parseBAR(s); }
    else if (token.equals(“b”))    { return parseBAZ(s); }
    else                           { return false;  }         // PARSE ERROR
}
```

nb: will both read token & move scanner…

# Top-Down Recursive Descent Parsers:

- Build a set of mutually-recursive procedures, based on the grammar.
- One procedure for each nonterminal.
- Choose which branch to follow based on next input token.
- Within branch, test for each terminal/nonterminal in turn.
- Fail if expected token is missing or no option available.
- Return Boolean if just checking, or parse tree.
- Must always able to choose next path given current state and next token!

example: "add(-5, sub(50, 50), 4)"

*don't* want to move scanner??

Note : next token is "add" so we want to enter the parseAdd() method, but parseAdd() itself will need to check the token "add", so… 😐 actually the code on previous page won't do.

# Java's scanner can "peek" at the next token

- Need to be able to look at the next token to work out which branch to take, *but not move the scanner along!*
  - Scanner has two forms of hasNext:
    - sc.hasNext():
      → is there another token in the scanner?
    - sc.hasNext("string to match"):
      → is there another token, and does it match the string?

```
if ( sc.hasNext("add") ) { …
```

  - Can use this to peek at the next token
  - String can be a regular expression!

```
if ( sc.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.

# Two kinds of rule, #1: rules with choice

THING ::= W1 | W2 | … | Wn

where each W is a sequence of terminal and/or nonterminal symbols

- Parser has to choose which W branch to take:

```
parseTHING {
    if next token can start W1 { parse W1 }
    else if next token can start W2 { parse W2 }
    …
    else fail
}
```

next token *can't* start a THING

# Two kinds of rule, #2: rules <u>without</u> choice

THANG ::= X1  X2  …  Xn

where each X is a terminal or nonterminal symbol.

- Parser has to check  all of X1,  X2,  … Xn, in turn.

```
parseTHANG {
    parse X1; … parse Xn;
}
```

- Fail if any can't be parsed.
- Note: Empty rules need a bit more care!

next lecture

# Parsing Expressions (checking only)

from grammar, Expr is “with choice” flavour

```
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;
}
```

Add is the other flavour

```
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)

Notice parseSub, parseMul, parseDiv are the *same as* parseAdd:

```
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;
}
```

# 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")) {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;
}
```

Notice this amounts to changing the grammar to:

Expr ::= Num | Op "(" Expr "," Expr ")"
Op ::= "add" | "sub" | "mul" | "div"
Num ::= [-+]?[0-9]+

(and writing the code for parseOP and parseNum inline)

# Simplifying the parser

We can reduce the duplication in checking for terminals

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

// consume next token and return true if it matches pat, else false

```
public String require(Scanner s, String pat,){
    if ( s.hasNext(pat) ) { s.next(); return true; }
    else { return null; }  // Print error message?
}
```

# Patterns are better

- patterns with names make program **easier to understand**
- patterns can be pre-compiled, for **efficiency**

```
Pattern numPat = Pattern.compile(
                "[-+]?(\\d+([.]\\d*)?|[.]\\d+)");
Pattern addPat = Pattern.compile("add");
Pattern subPat = Pattern.compile("sub");
Pattern mulPat = Pattern.compile("mul");
Pattern divPat = Pattern.compile("div");
Pattern opPat = Pattern.compile("add|sub|mul|div");
Pattern openPat = Pattern.compile("\\(");
Pattern commaPat = Pattern.compile(",");
Pattern closePat = Pattern.compile("\\)");
// Should all be declared as private and final.
```

# Patterns are better

```
public Node parseExpr(Scanner s) {
   Node n;
   if (!s.hasNext())         { return false; }
   if (s.hasNext(numPat)) { return parseNumber(s); }
   if (s.hasNext(addPat)) { return parseAdd(s); }
   if (s.hasNext(subPat)) { return parseSub(s); }
   if (s.hasNext(mulPat)) { return parseMul(s); }
   if (s.hasNext(divPat)) { return parseDiv(s); }
   return false;
}
```

# Constructing a full parse tree (ie. not just checking input)

- Given our pet 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]+

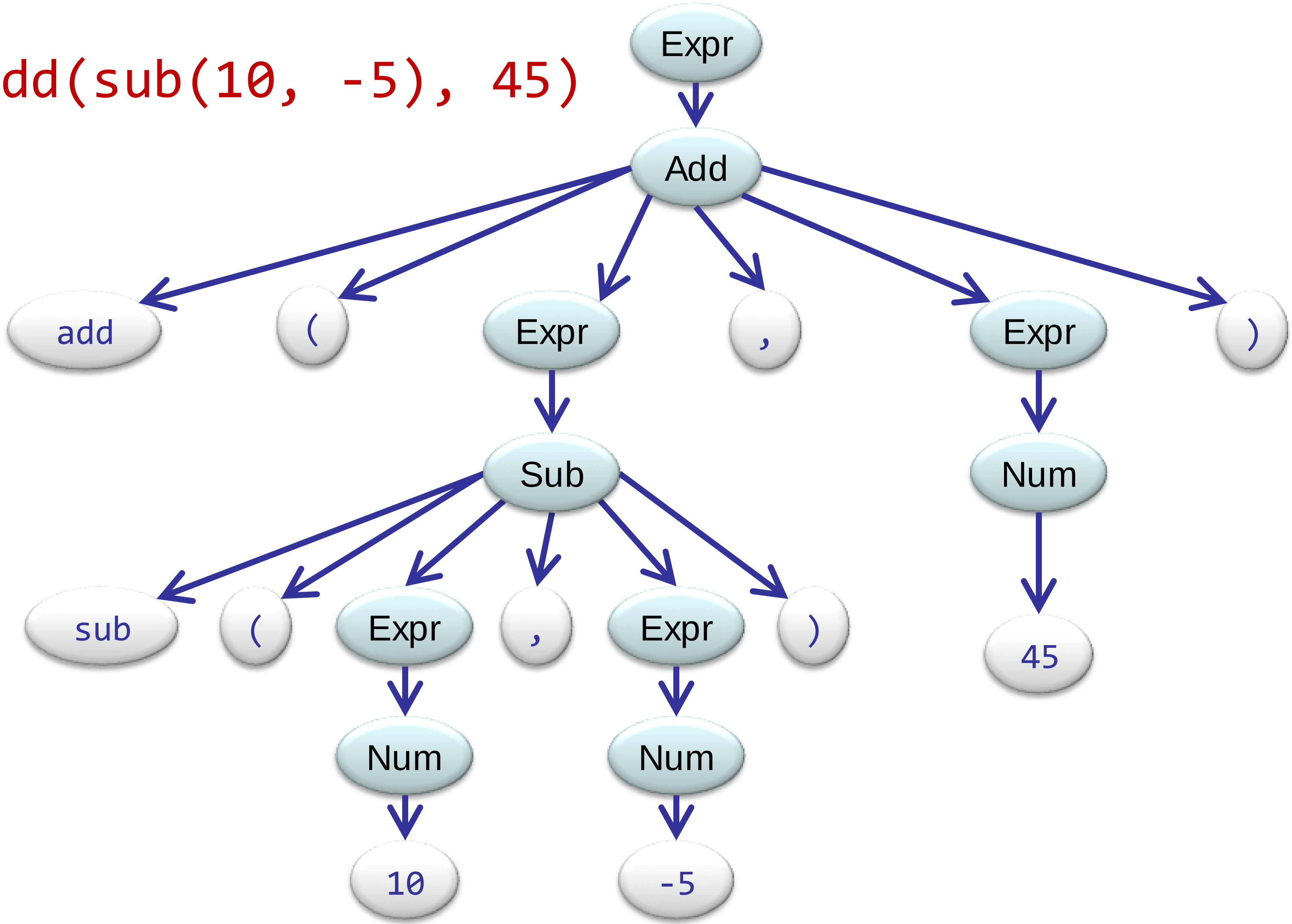
- And an expression:

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

- How can we construct a parse tree?

# Constructing a parse tree

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



# Building a parse tree

Each parse method returns a parse tree, rather than a Boolean.

```
public Node parseExpr(Scanner sc)

public Node parseNum(Scanner sc)

public Node parseAddNode(Scanner sc)

public Node parseSubNode(Scanner sc)

public Node parseMulNode(Scanner sc)

public Node parseDivNode(Scanner sc)
```

# Data structure for parse tree:   two options

1. Use different node type for each kind of expression:
   - Expression node
     - Contains a Number or an Add/Sub/Mul/Div
   - Add, Sub, Mul, Div node
     - Contains the operator, “(“, first expression, “,”, second expression, and “)”
   - Number node
     - Contains a number
   - Terminal node
     - Contains a string

We’ll do 1.
Ex: Adapt to do 2

2. Use general tree, with label at each node and list of children.

# Data structure for parse tree

```
interface Node { }

class ExprNode implements Node {
   final Node child;
   public ExprNode(Node ch){ child = ch; }
   public String toString() { return "[" + child + "]"; }  //
   Brackets added to show structure.
}

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; }
}
```

# Data structure for parse tree

```
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 + "]";
   }
}

// SubNode, MulNode and DivNode similar
```

# Handling errors

- Can't return false to indicate parse failure.
- Let's make the parser **throw an exception** if there is an error!
  - so each method *either* returns a valid Node, *or* throws an exception.
  - this `fail` method throws exception, with message providing 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);
}
```

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

# Building a parse tree

Suggestion: collect the components, then build the required node.

```
public Node parseExpr(Scanner s) {
   if (!s.hasNext())                { fail("Empty expr",s); }
   Node child = null;
   if (s.hasNext("-?\\d+"))         { child = parseNum(s);}
   else if (s.hasNext("add"))       { child = parseAdd(s); }
   else if (s.hasNext("sub"))       { child = parseSub(s); }
   else if (s.hasNext("mul"))       { child = parseMul(s); }
   else if (s.hasNext("div"))       { child = parseDiv(s); }
   else { fail("not an expression", s); }
   return new ExprNode(child);
}
```

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

# Building a parse tree

```
public Node parseAdd(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);
}
```

we don’t need to check whether parse methods succeed!

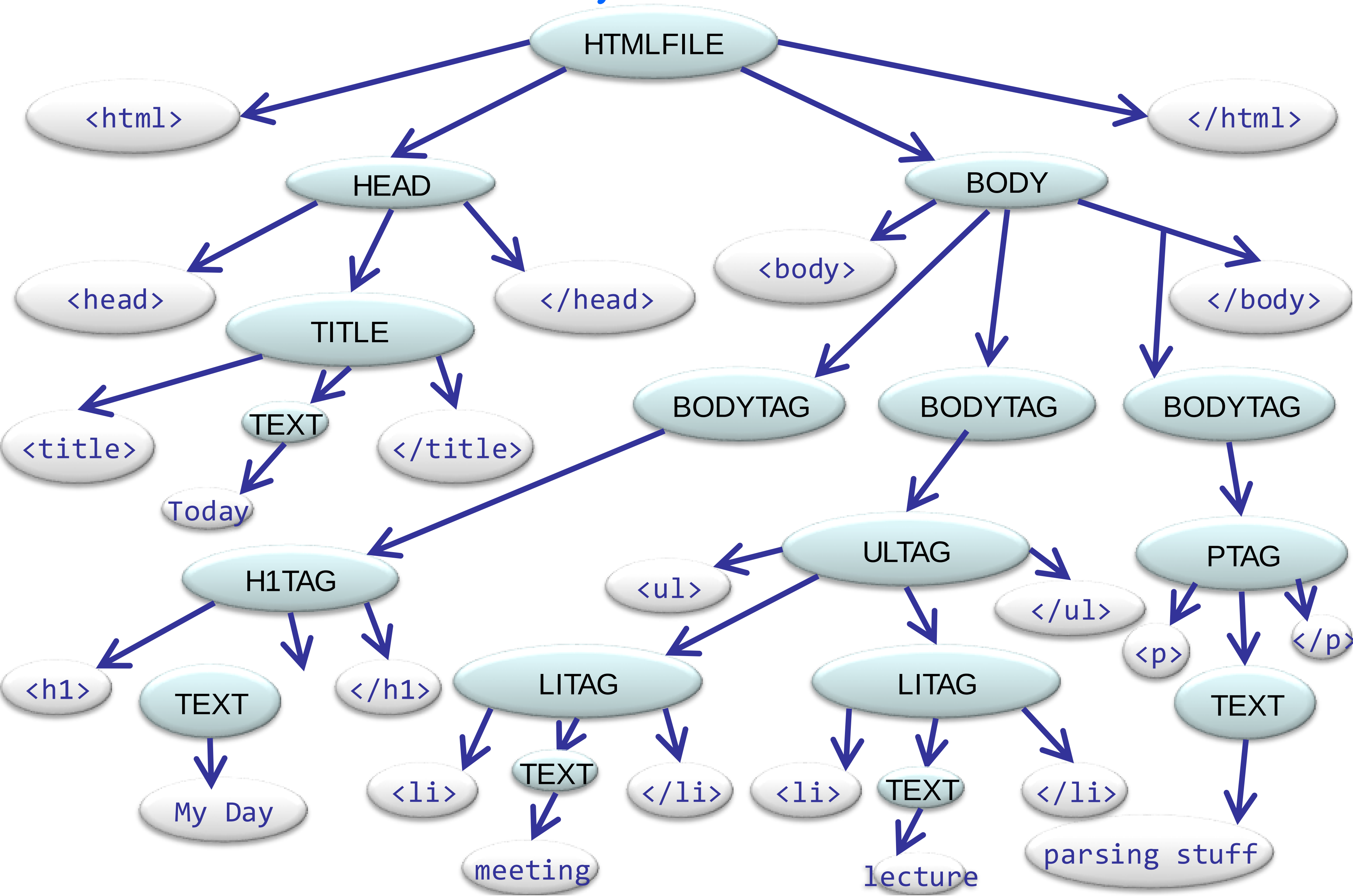
# Is that too much information?

Concrete syntax trees contain a lot of redundant information.

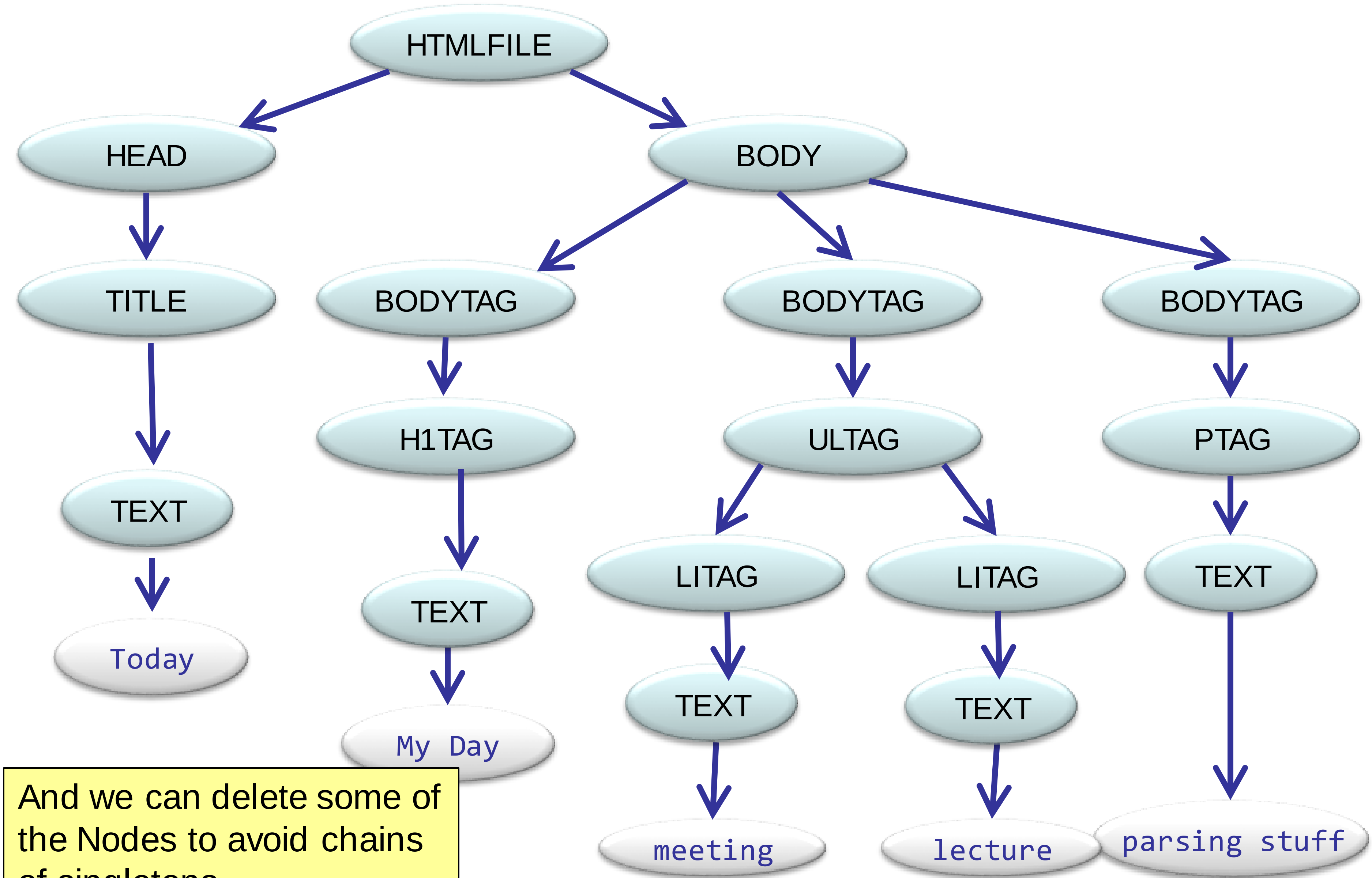
E.g. in parse tree for html file, we know that every HEAD has “<head>” and “</head>” terminals. We only care about what TITLE there is and only the unknown string part of the title.

- An **abstract syntax tree (AST)** represents the abstract syntactic structure of the text.
- Each node of the tree denotes a construct occurring in the text.
- The syntax is ‘abstract’ in that it does not represent all the elements of the full syntax.
- Only keep things that are semantically meaningful.

# Recall the Concrete Syntax Tree :



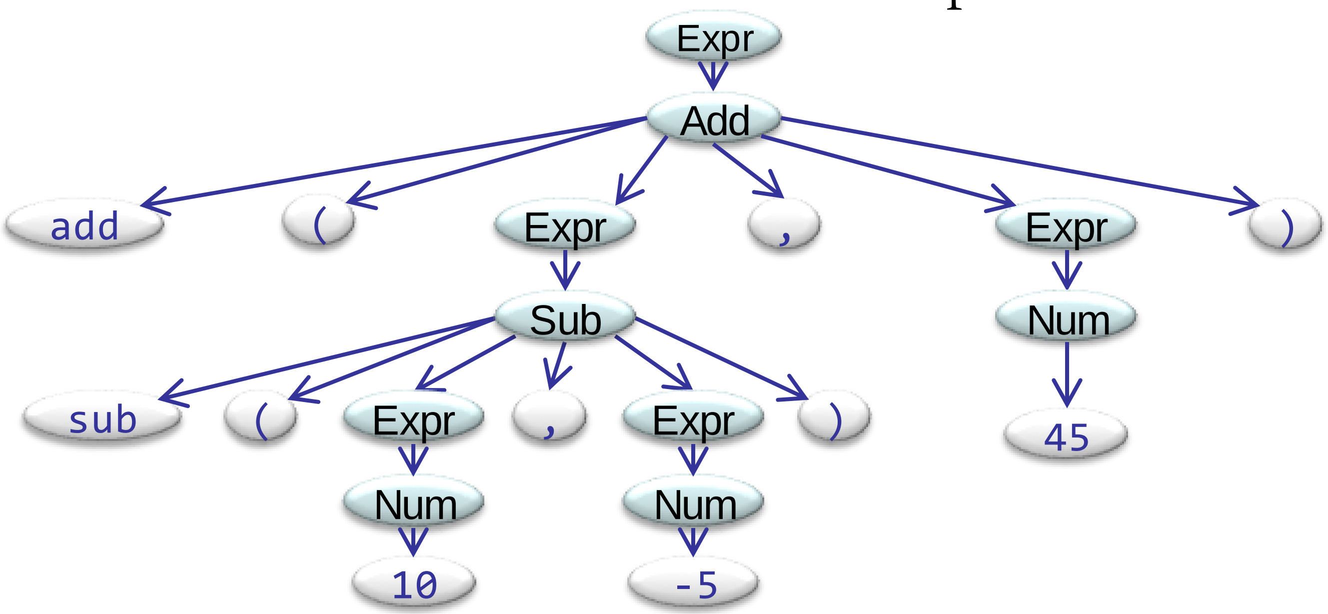
# versus the Abstract Syntax Tree (AST)



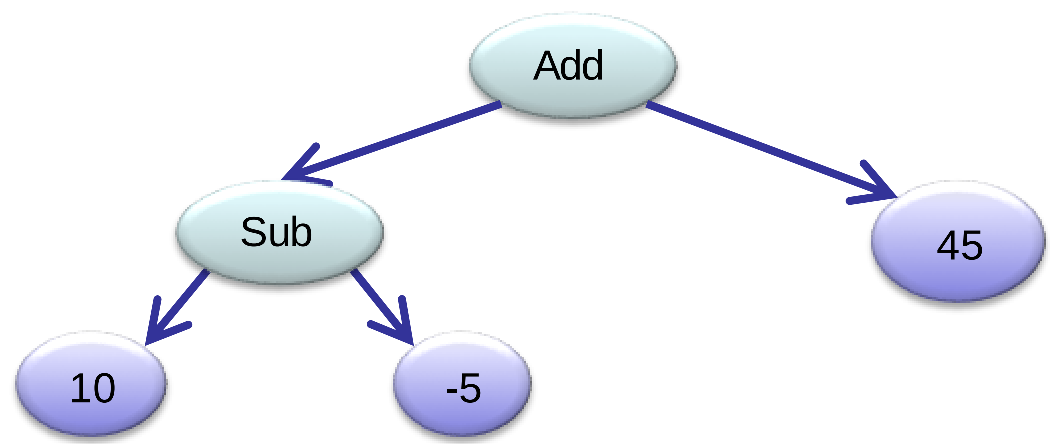

And we can delete some of the Nodes to avoid chains of singletons

# Abstract Syntax Trees for arithmetic expressions

- We don’t 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



# Data structure for ASTs

Don't need ExprNode or TerminalNode

NumNode stays the same:

---

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

# Data structure for ASTs

`AddNode` gets simpler:

Only need the two arguments

```
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+")";}
}
```

# *c.f.* the old AddNode class (for *concrete* tree)

from a few slides back….

```
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 + "]";
   }
}
```

# Building an AST: the parse methods

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

# Building an AST: the parse methods

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 parseNum(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;
}
```

# Building an AST: the parse methods

parseAdd *etc* are simpler – yay!

```
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);
}
```

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

# What can we do with an AST?

- **We can "execute" parse trees!**

```
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 an 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 + " + " + right + ")";
   }
}
```

Print in the human-friendly “infix” notation (with brackets)