COMP261 Lecture 18

Lindsay Groves

Parsing 3 of 4:
Constructing parse trees, error handling

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

Recursive Descent Parsing - recap

- 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!

Recursive Descent Parsing

For basic grammars, we have two kinds of rule:

- 1: Rules with choice:

  N ::= W1 | W2 | … | Wn (n > 1)

  where wach Wi is a sequence of terminal and/or nonterminal symbols.

- Parser has to choose which branch to take:

```
parseN {
  if next token can start W1 { parse W1 }
  else if next token can start W2 { parse W2 }
  …
  else fail(next token can't start N);
}
```

Recursive Descent Parsing

- 2: Rules without choice:

  N ::= X1 X2 … Xn

  where each Xi is a terminal or nonterminal symbol.

- Parser looks for X1, X2, … Xn, in turn.

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

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

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")) {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;
}
```

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?
}
```

A Better parser: using patterns

- Give names to patterns to make program easier to understand and to modify
- Precompile the patterns 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.
```

A Better parser: using patterns

```
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 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]+

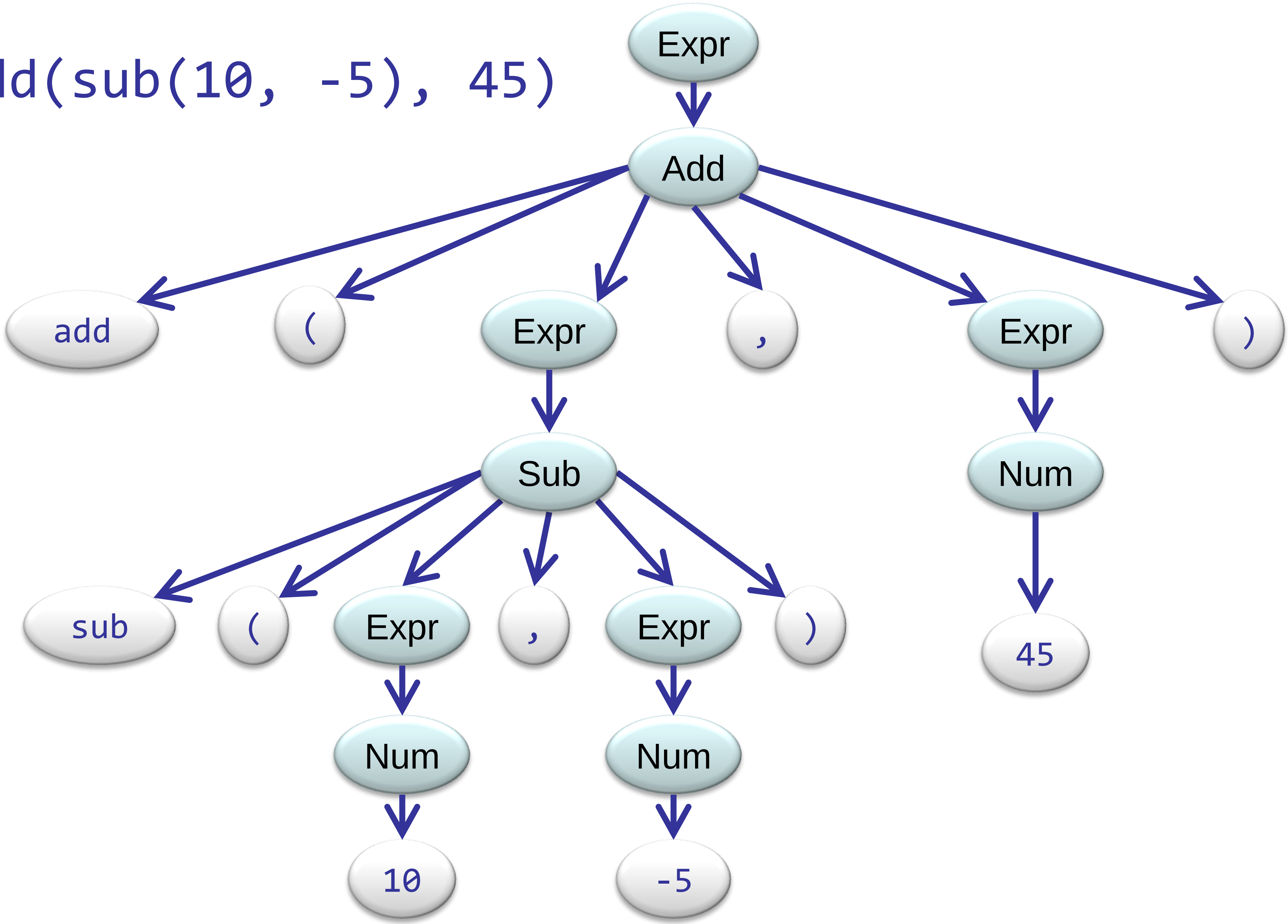
- And an expression:

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

- 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 s)

public Node parseNum(Scanner s)

public Node parseAddNode(Scanner s)

public Node parseSubNode(Scanner s)

public Node parseMulNode(Scanner s)

public Node parseDivNode(Scanner s)
```

Data structure for parse tree

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 expressin, and “)”
   - Number node
     - Contains a number
   - Terminal node
     - Contains a string
2. Use a general tree structure with a label at each node and list of children.

We’ll do 1.
Ex: Adapt to do 2

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.
- Could return null, or add an "Error" node .
- Or make the parser throw an exception if there is an error:
  - each method either returns a valid Node, or 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 (

Building a parse tree

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);
}
```

Don’t need to check whether parse methods succeed!

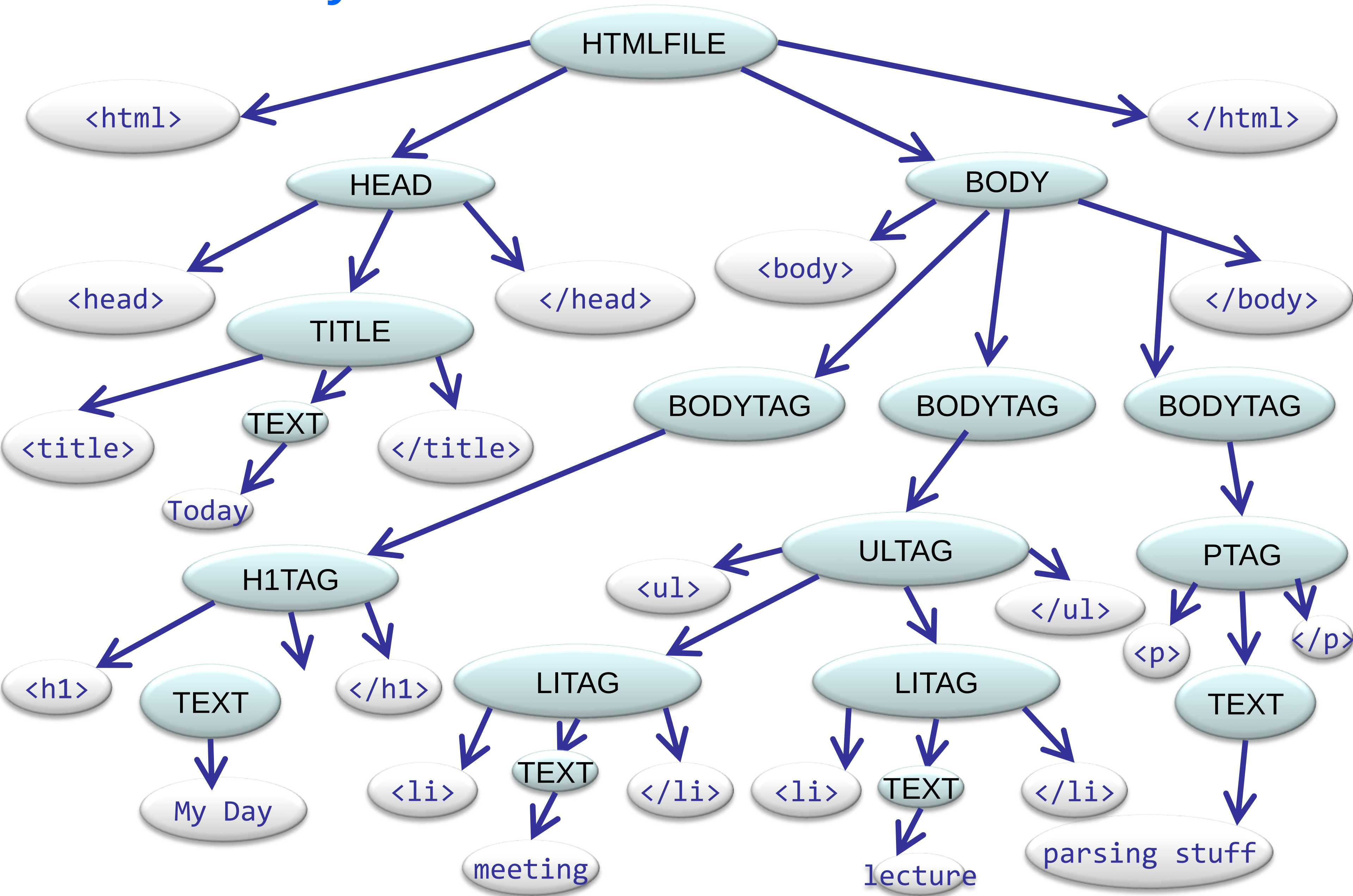
Is that too much information?

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.

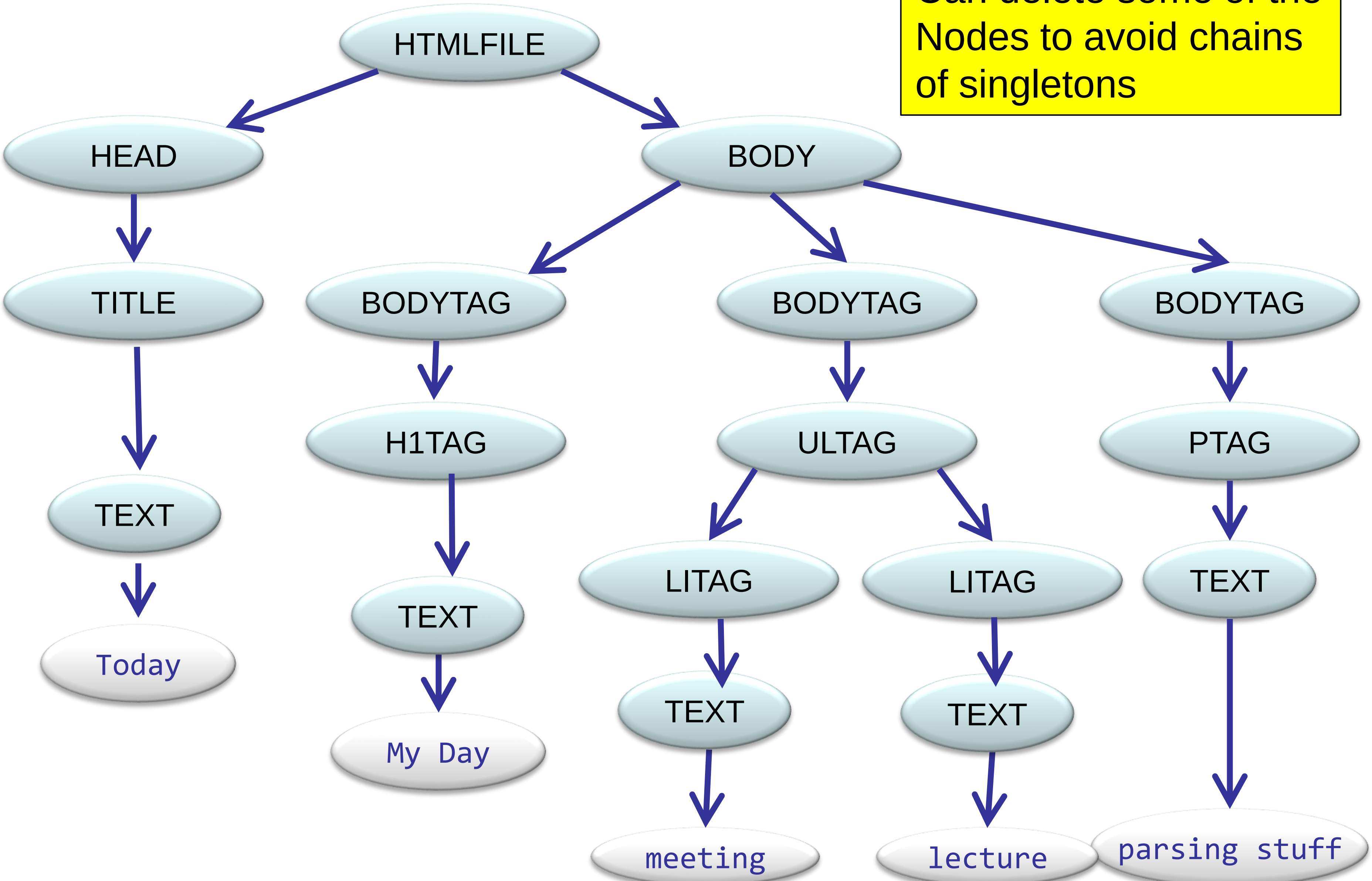
- An **abstract syntax tree (AST)** represents the abstract syntactic structure of the text.
- Each node of the tree denotes a construct occurring in the text.
- The syntax is ‘abstract’ in that it does not represent all the elements of the full syntax.
- Only keep things that are semantically meaningful.

Abstract Syntax Tree :



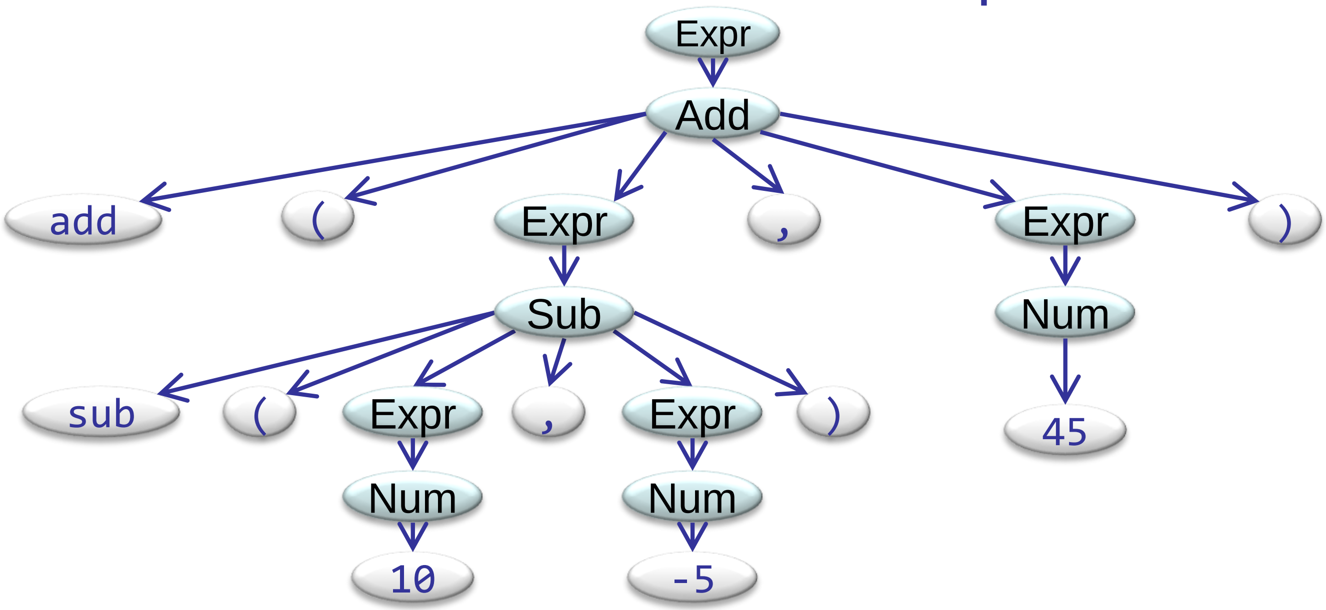
Abstract Syntax Tree (AST)

Can delete some of the Nodes to avoid chains of singletons

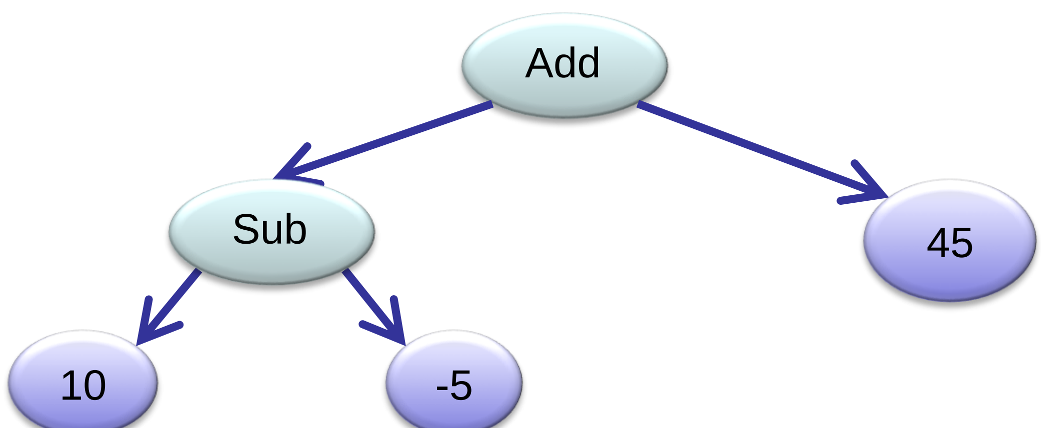


Abstract syntax trees for arithmetic expressions

- 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 is 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+")";}
}
```

Building an AST

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

Building an AST

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

parseAdd *etc* are 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);
}
```

```
// consume (and return) next token if it matches pat, report 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() + rig
  }
  public String toString() {
    return "(" + left + " + " + right + ")";
  }
}
```

Print in regular infix notation (with brackets)

Extending the 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; }
}
```

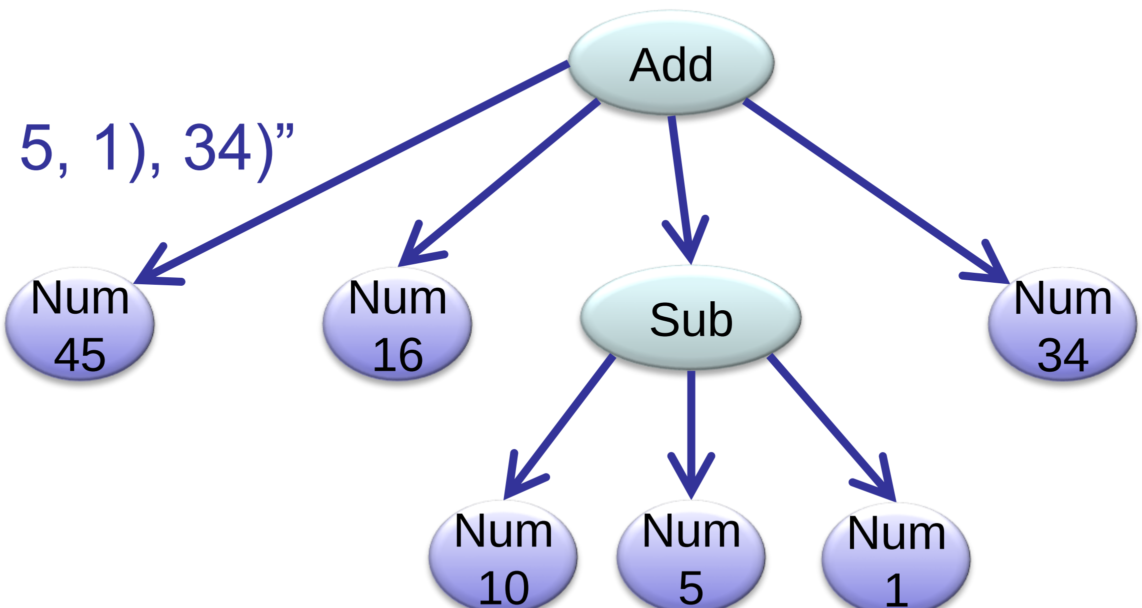
Extending the Language

- We can 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)”



Extending the 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; }
}
```

Extending the 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;
    }
```