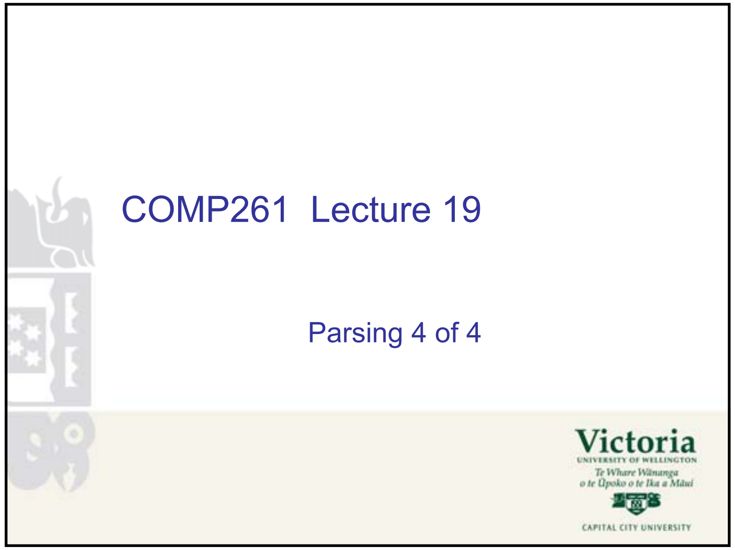
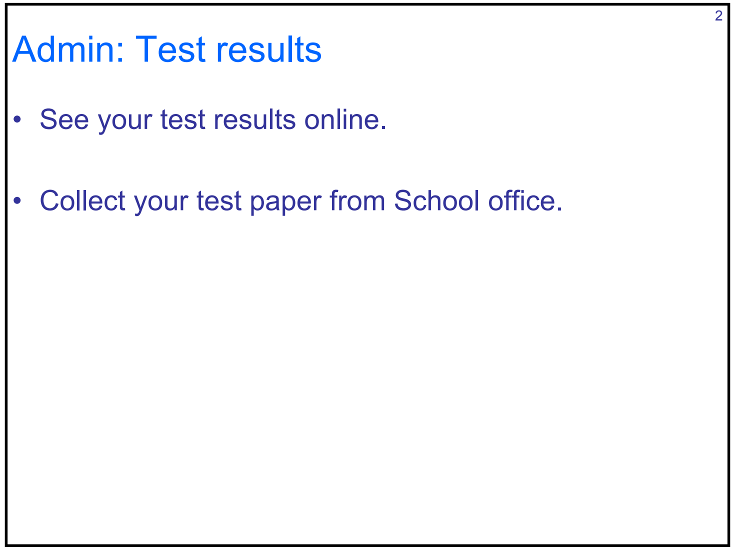


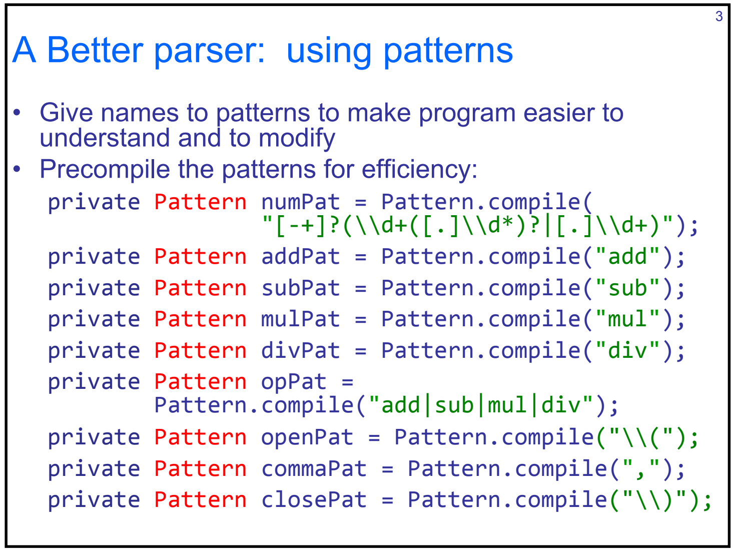
COMP261 Lecture 19

Parsing 4 of 4



Admin: Test results

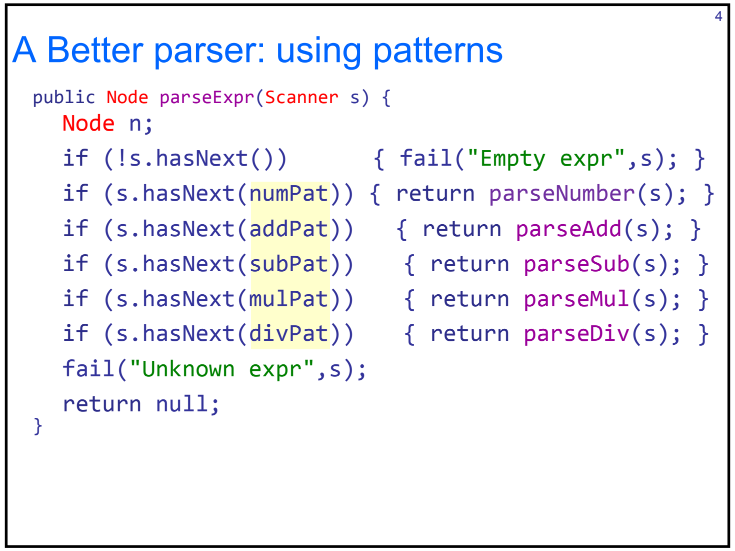
- See your test results online.
- Collect your test paper from School office.



A Better parser: using patterns

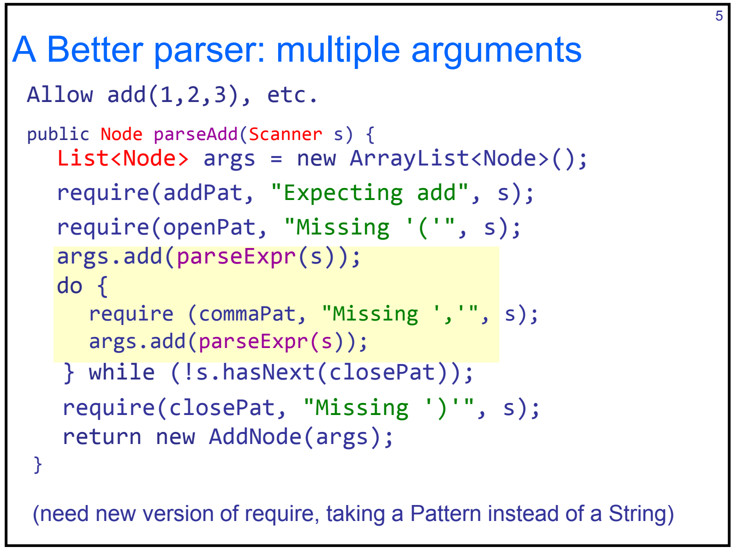
- Give names to patterns to make program easier to understand and to modify
- Precompile the patterns for efficiency:

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



A Better parser: using patterns

```
public Node parseExpr(Scanner s) {
  Node n;
  if (!s.hasNext())      { fail("Empty expr",s); }
  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); }
  fail("Unknown expr",s);
  return null;
}
```

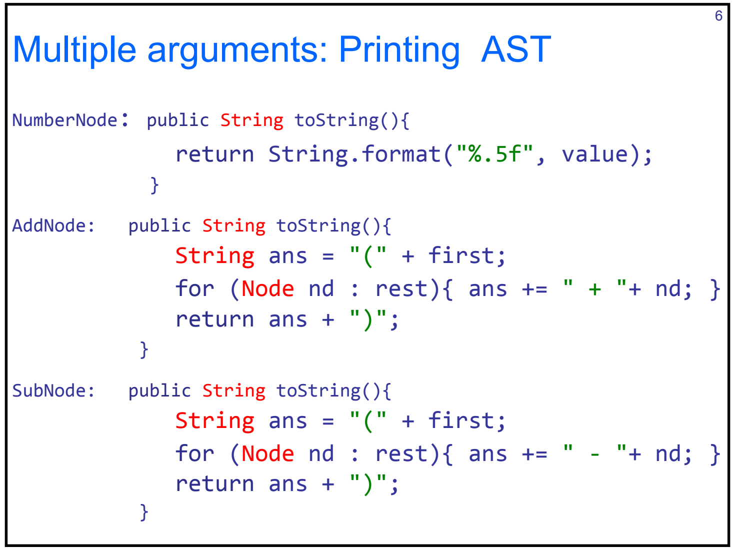


A Better parser: multiple arguments

```
Allow add(1,2,3), etc.

public Node parseAdd(Scanner s) {
  List<Node> args = new ArrayList<Node>();
  require(addPat, "Expecting add", s);
  require(openPat, "Missing '('", s);
  args.add(parseExpr(s));
  do {
     require (commaPat, "Missing ','", s);
     args.add(parseExpr(s));
   } while (!s.hasNext(closePat));
   require(closePat, "Missing ')'", s);
   return new AddNode(args);
}
```

(need new version of require, taking a Pattern instead of a String)

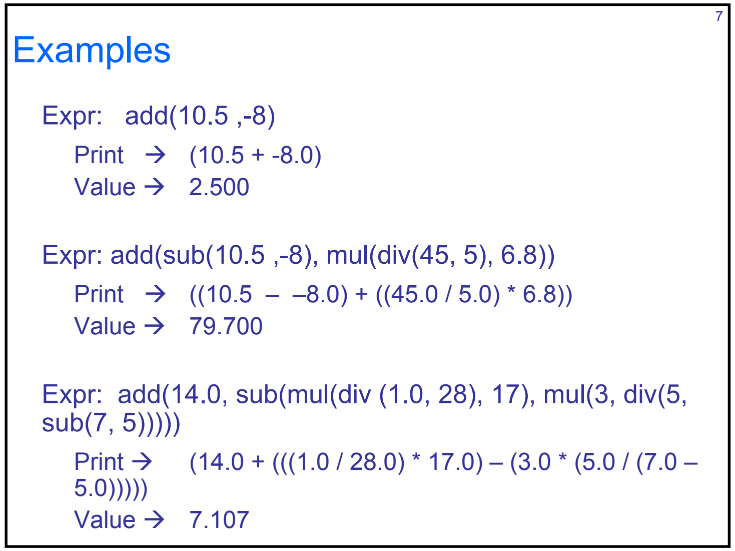


Multiple arguments: Printing AST

```
NumberNode: public String toString(){
               return String.format("%.5f", value);
             }

AddNode:    public String toString(){
               String ans = "(" + first;
               for (Node nd : rest){ ans += " + "+ nd; }
               return ans + ")";
            }

SubNode:    public String toString(){
               String ans = "(" + first;
               for (Node nd : rest){ ans += " - "+ nd; }
               return ans + ")";
            }
```



Examples

Expr: add(10.5 ,-8)

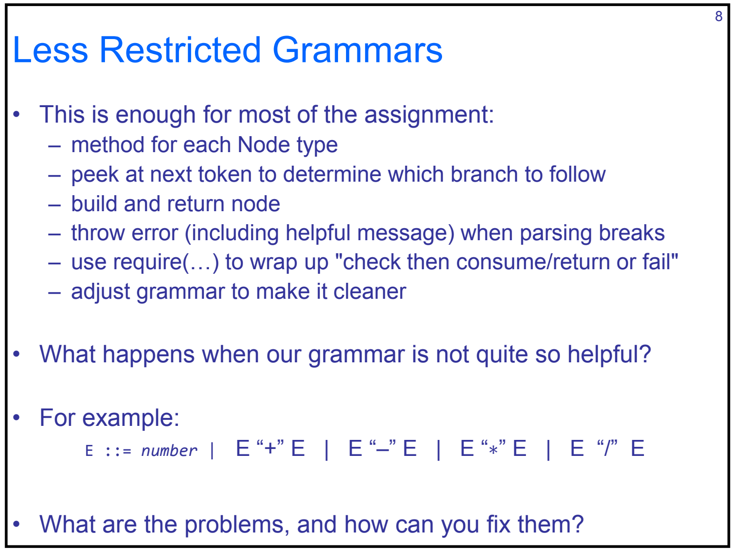
Print → (10.5 + -8.0)
Value → 2.500

Expr: add(sub(10.5 ,-8), mul(div(45, 5), 6.8))

Print → ((10.5 – –8.0) + ((45.0 / 5.0) * 6.8))
Value → 79.700

Expr: add(14.0, sub(mul(div (1.0, 28), 17), mul(3, div(5, sub(7, 5)))))

Print → (14.0 + (((1.0 / 28.0) * 17.0) – (3.0 * (5.0 / (7.0 – 5.0)))))
Value → 7.107



Less Restricted Grammars

- This is enough for most of the assignment:
  - method for each Node type
  - peek at next token to determine which branch to follow
  - build and return node
  - throw error (including helpful message) when parsing breaks
  - use require(…) to wrap up "check then consume/return or fail"
  - adjust grammar to make it cleaner
- What happens when our grammar is not quite so helpful?
- For example:

  `E ::= number` | E "+" E | E "–" E | E "*" E | E "/" E

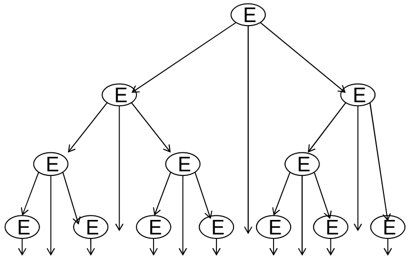
- What are the problems, and how can you fix them?

9

Ambiguous Grammars

Grammar:

E ::= *number* | E “+” E | E “–” E | E “∗” E | E “/” E



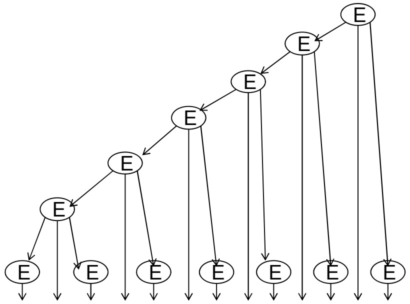
Text: 65 * 74 – 68 + 25 * 5 / 3 + 16

10

Possible Parses

Grammar:

E ::= *number* | E “+” E | E “–” E | E “∗” E | E “/” E



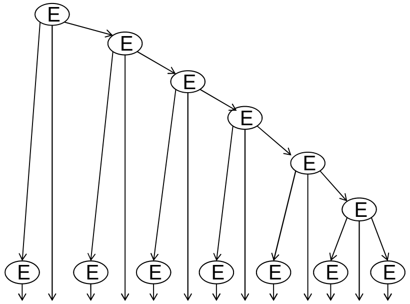
65 * 74 – 68 + 25 * 5 / 3 + 16

11

Possible Parses

Grammar:

E ::= *number* | E “+” E | E “–” E | E “∗” E | E “/” E



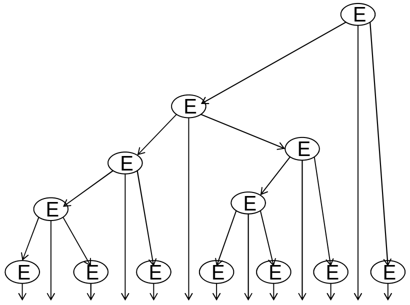
65 * 74 – 68 + 25 * 5 / 3 + 16

12

Possible Parses

Grammar:

E ::= *number* | E “+” E | E “–” E | E “∗” E | E “/” E

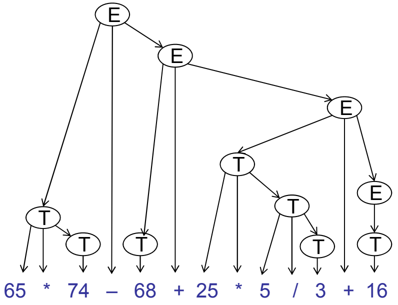


65 * 74 – 68 + 25 * 5 / 3 + 16

Ambiguous Grammars

- If a grammar allows multiple parses then we need to specify which we want (if it makes a difference)
- For example:

  EXPR ::= TERM | TERM “+” EXPR | TERM “–” EXPR
  TERM ::= *number* | *number* “∗” TERM | *number* “/” TERM



Telling which option to follow

EXPR ::= TERM | TERM “+” EXPR | TERM “–” EXPR
TERM ::= *number* | *number* “∗” TERM | *number* “/” TERM

- Break into subrules, collecting the shared elements:

  EXPR ::= TERM RESTOFEXPR
  RESTOFEXPR ::= “+” EXPR | “–” EXPR | ∊

  TERM ::= *number* RESTOFTERM
  RESTOFTERM ::= “∗” TERM | “/” TERM | ∊

  (∊ means “empty string”)

- Transformations such as these can often turn a problematic grammar into a tractable grammar

A more practical approach

- Instead of

  E ::= *number* | E “+” E | E “–” E | E “∗” E | E “/” E

- Write:

  E ::= *number* [Op *number*]*
  Op := “+” | “–” | “∗” | “/”

- And the parser as:

```
parseNum(s);
while (!s.hasNext(opPat)) {
  s.next();
  parseNum(s);
}
```

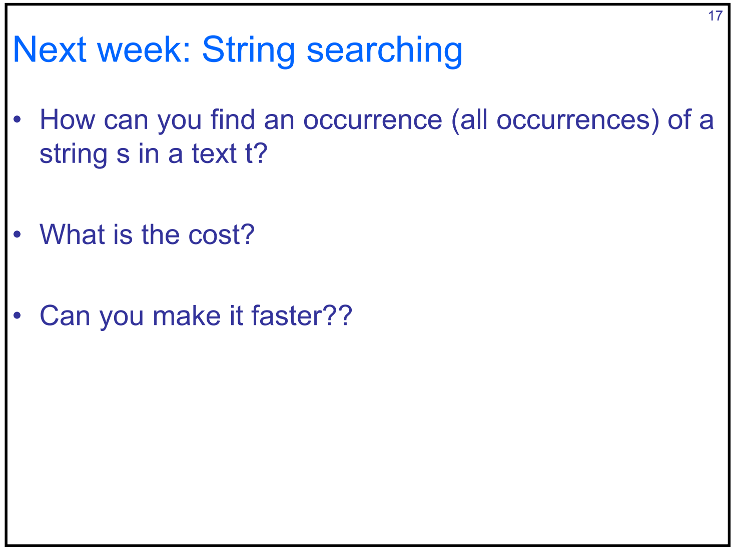
A more practical approach

- What about operator precedence: * before +, etc “
- Grammar:

  E ::= T [(“+”|”-”) T]* Expression
  T ::= F [(“*”|”/”) F]* Term
  F ::= number | “(“ E “)” Factor

- Parser:

```
public parseE(s) {
  parseT;
  while (!s.hasNext(addOpPat)) { // + or -
    s.next();
    parseT(s);
}
```



17

Next week: String searching

- How can you find an occurrence (all occurrences) of a string s in a text t?
- What is the cost?
- Can you make it faster??