Lecture 14:

- Associativity and Precedence

Announcements:

- HW-3 out (due Wed)

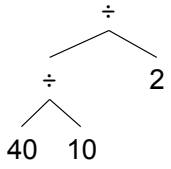
More on Context Free Grammars

Operator associativity and precedence under context-free grammars

- both are tricky ... we will discuss approaches (not needed for HW-3)

Operator associativity ... chaining together same op

- most operators are **left associative** ... e.g., $\times$, $\div$, $+$, $-$
- For example ... $40 \div 10 \div 2 \equiv (40 \div 10) \div 2$



- Captured by the left-recursive (!) rule: $e \rightarrow e \div n$ ... so not $LL(k)$
- Bigger impact for non-commutative operations (where $a \circ b \not\equiv b \circ a$)

(*) % (mod), ** (exp), = (in C/C++) are examples of right-associative ops

Left Associativty under $LL(k)$

(1) One approach is to **rewrite the AST** after parsing

- similar to applying rotations in balanced-tree data structures (e.g., AVL)

(2) Another is to **modify** the grammar and parser ... to build the correct AST

- for **left-associative** ops use iteration via (Kleene) star
- for **right-associative** ops use (tail) recursion (natural for recursive descent)

Left Associativty under $LL(k)$

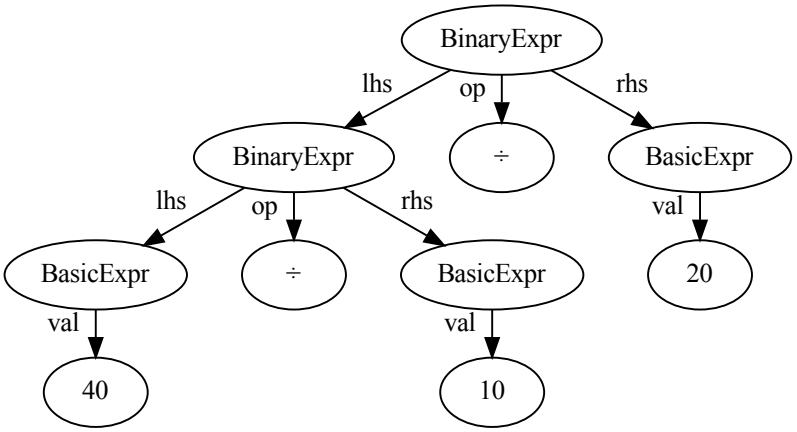
Example:

- Modified grammar rule: $e \rightarrow n\ (\ \div\ n\ )^*$ ... for n an `INT_VAL`
- Modified parser: ... assuming `Expr`, `Basic`, and `Binary` classes

```
public Expr expr() {
  BasicExpr valNode = new BasicExpr();
  valNode.val = currToken();
  eat(TokenType.INT_VAL, "...");
  Expr node = valNode;
  while (match(TokenType.DIVIDE)) {
    BinaryExpr opNode = new BinaryExpr();
    opNode.lhs = node;
    opNode.op = currToken;
    advance();
    BasicExpr rhsNode = new BasicExpr();
    rhsNode.val = currToken;
    eat(TokenType.INT_VAL, "...");
    opNode.rhs = rhsNode;
    node = opNode;
  }
  return node;
}
```

Left Associativty under $LL(k)$

Check In: Trace code and build the AST for "40 ÷ 10 ÷ 20" ...



Operator Precedence

Operator precedence ... how to evaluate mixed-operator expressions

- e.g., division (÷) has higher precedence than addition (+)

  `2 + 3 ÷ 4` ≡ `2 + (3 ÷ 4)`
  `2 ÷ 3 + 4` ≡ `(2 ÷ 3) + 4`

One Solution: Encode precedence in the grammar ...

$e \rightarrow t\ (\ +\ t\ )^*$ ... group non-+ part as t

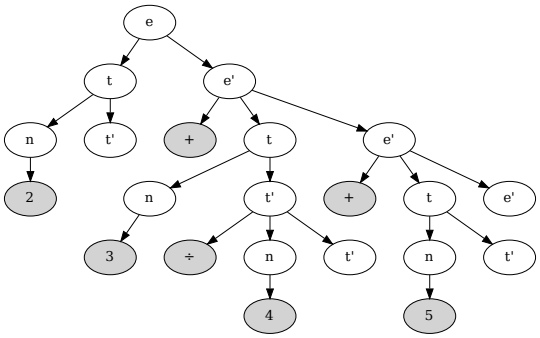
$t \rightarrow n\ (\ \div\ n\ )^*$ ... normal (left-assoc) ÷ rule

Operator Precedence

Slightly Modified Version: ... for the parse tree

$$e \rightarrow t\ e'$$
$$e' \rightarrow +\ t\ e' \mid \varepsilon$$
$$t \rightarrow n\ t'$$
$$t' \rightarrow \div\ n\ t' \mid \varepsilon$$

Check In: Draw the parse tree for: 2 + 3 ÷ 4 + 5



If you look hard enough, can see the correct grouping: 2 + (3 ÷ 4) + 5