Lecture 16:

- Associativity and Precedence

Announcements:

- HW-3 out
- Quiz 4 on Friday (AST creation, visitors)

More on Context Free Grammars

With recursive descent parsers, it can be hard to ...

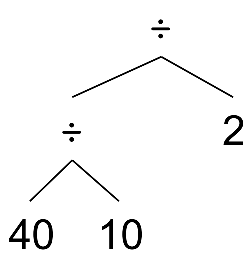
- define grammars with appropriate operator **associativity**
- define grammars with appropriate operator **precedence**
- ... and these are important for semantic analysis (and evaluation)

Operator associativity

- many operators are **left associative** ... e.g., $\times$, $\div$, $+$, $-$
- For example ... $40 \div 10 \div 2 \equiv (40 \div 10) \div 2$
- Can be captured by the grammar rule:

  $e \rightarrow e \div n$ ... n a number

- and the "AST":



- But notice this requires ulineleft recursion! ... so not $LL(k)$

Dealing with left-associative operators

- One approach is to **rewrite the AST** after parsing
  - similar to applying rotations in Red-Black or AVL trees

- Another is to **modify** the grammar and recursive-descent parser
- ... to construct the correct AST

Example:

The grammar rule (modified to be $LL(k)$) ...

$$e \rightarrow n \text{ (} \texttt{DIVIDE} \; n \text{)}^*$$

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

Modify the "normal" recursive descent function to build left-associative AST ...

```python
def e(self):
    expr_node = ValExpr(val=self.curr_token)
    self.eat(TokenType.INT_VAL, '...')
    while self.match(TokenType.DIVIDE):
        self.advance()
        sub_expr_node = ValExpr(val=self.curr_token)
        self.eat(Token.INT_VAL, '...')
        tmp = DivExpr(lhs=expr_node, rhs=sub_expr_node)
        expr_node = tmp
    return expr_node
```

Check In: Trace the code above and show the AST for $40 \div 10 \div 2$.

The result is:

```
expr_node = ValExpr(40)

sub_expr_node = ValExpr(10),
expr_node = DivExpr(ValExpr(40), ValExpr(10))

sub_expr_node = ValExpr(2),
expr_node = DivExpr(DivExpr(ValExpr(40), ValExpr(10)), ValExpr(2))
```

Operator precedence

- Division (/) has higher precedence than addition (+)
- For example:

  `2 + 3 / 4` $\equiv$ `2 + (3 / 4)`

  `2 / 3 + 4` $\equiv$ `(2 / 3) + 4`

One solution: Encode precedence in the grammar

$e \rightarrow t$ ($\texttt{PLUS}$ t)*

$t \rightarrow \texttt{INT}$ ($\texttt{DIVIDE}$ $\texttt{INT}$)*

- This is equivalent to ...

  $e \rightarrow t\ e'$

  $e' \rightarrow \texttt{PLUS}\ t\ e' \mid \epsilon$

  $t \rightarrow \texttt{INT}\ t'$

  $t' \rightarrow \texttt{DIVIDE}\ \texttt{INT}\ t' \mid \epsilon$

Exercise: Draw the parse tree for: `2 + 3 / 4 + 5`

⋆ Don't need to consider associativity and precedence for HW-3

- but you should understand the issues and how to resolve them
- note it would be a good extension project

Summary – Things to Know

1. Difference between operator associativity and precedence.

2. The issue/challenge with encoding associativity into a grammar.

3. Options for "dealing with" associativity in a recursive descent parser.

4. Given an example, generate a recursive descent function that correctly builds a left-associative AST.

5. General approach for encoding precedence into a grammar.

6. Given an example, create a grammar that correctly encodes precedence.