Today

- More on ASTs
- Static Analysis

Announcements

- HW-4 out
- EX-4 out
- Exam-1 next Tuesday
- Quiz-4 Thursday: recursive descent
More on Context Free Grammars

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

- define grammars with appropriate operator \textit{associativity}
- define grammars with appropriate operator \textit{precedence}
- ... and these are important for semantic analysis (and evaluation)

Operator \textit{associativity}

- many operators are \textit{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:

$$e \rightarrow e \div n$$

- and the “AST”:

$$\begin{array}{c}
\div \\
\div \ \ 2 \\
\ \ 40 \ 10
\end{array}$$

- But notice this requires \textit{left 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:

\[ e \rightarrow \text{val} \ ( \div \ \text{val} )^* \]

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

```java
private DivExpr e() {
    ValExpr v = new ValExpr();
    v.val = currToken;  // assuming ValExpr holds a "value"
    Expr v1 = v;
    eat(VAL, "...");
    while (match(DIVIDE)) {
        advance();
        ValExpr v2 = new ValExpr();
        v2.val = currToken;
        eat(VAL, "...");
        DivExpr tmp = new DivExpr();  // assuming a lhs and rhs
        tmp.lhs = v1;
        tmp.rhs = v2;
        v1 = tmp;
    }
    return v1;
}
```
Exercise: Trace the code above and show the AST for $40 \div 10 \div 2$.

The result is:

$$v_1 = \text{ValExpr(val=40)}$$

$$v_2 = \text{ValExpr(val=10)},$$

$$v_1 = \text{DivExpr(lhs=ValExpr(val=40), rhs=ValExpr(val=10))}$$

$$v_2 = \text{ValExpr(2)},$$

$$v_1 = \text{DivExpr(lhs=DivExpr(lhs=ValExpr(val=40), rhs=ValExpr(val=10)), rhs=ValExpr(val=2))}$$
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**

\[
e \rightarrow t \ ( \ PLUS \ t \ )^* \\
t \rightarrow \ INT \ ( \ DIVIDE \ INT \ )^* 
\]

- This is equivalent to ...

\[
e \rightarrow t \ e' \ \\
e' \rightarrow PLUS \ t \ e' \mid \epsilon \\
t \rightarrow \ INT \ t' \\
t' \rightarrow DIVIDE \ INT \ t' \mid \epsilon 
\]

**Exercise:** Draw the parse tree for: \( 2 + 3 / 4 + 5 \)

* Don’t need to consider associativity and precedence for HW-4
  - but you should understand the issues and how to resolve them
  - note it would be a good extension project