Today

- Operator Associativity & Precedence
- AST Navigation

Assignments

- HW4 out (due next Thurs)
Generating Abstract Syntax Trees (ASTs)

1. The parsing step both checks syntax and builds the AST

2. An AST is typically used for:
   - semantic analysis, e.g., type checking, ensuring items defined before used
   - interpretation, e.g., in an AST interpreter
   - conversion to intermediate representation (like bytecode)

3. An AST is like an “expression tree” ...

- perform “in-order traversal” (left, node, right) to “execute” expression tree
- more types of nodes in an AST, e.g., declarations, loops, var assignment, etc.
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 (e.g., evaluation)

Operator **associativity**

- many operators are *left associative* ... e.g., ×, ÷, +, −
- For example ... $40 ÷ 10 ÷ 2 ≡ (40 ÷ 10) ÷ 2$
- Can be captured by the grammar rule:

  $e \rightarrow e ÷ n$

- and the “AST”:

  40 10
      ÷
    2

  \( \frac{40}{10} ÷ 2 \)

- But notice this requires *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:

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

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

```python
def __e(self):
    v1 = ast.ValExpr()
    v1.val = self.current_token
    self.__eat(token.VAL, '...')
    while self.current_token.tokentype == token.DIVIDE:
        self.__advance()
        v2 = ast.ValExpr()
        v2.val = self.current_token
        self.__eat(token.VAL, '...')
        tmp = ast.DivExpr()
        tmp.left_operand = v1
        tmp.right_operand = v2
        v1 = tmp
    return v1
```
**Exercise**: Trace the code above and show the AST for $40 \div 10 \div 2$.

The result is:

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

$$v_2 = \text{ValExpr}(\text{val}=10),$$
$$v_1 = \text{DivExpr}(\text{lhs}=\text{ValExpr}(\text{val}=40), \text{rhs}=\text{ValExpr}(\text{val}=10))$$

$$v_2 = \text{ValExpr}(2),$$
$$v_1 = \text{DivExpr}(\text{DivExpr}(\text{lhs}=\text{ValExpr}(\text{val}=40), \text{ValExpr}(\text{val}=10)), \text{ValExpr}(\text{val}=2))$$
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

\[
e \rightarrow t ( '+' t )* \\
t \rightarrow num ( '/ ' num )* \\
\]

- This is equivalent to ...

\[
e \rightarrow t e' \\
e' \rightarrow '+' t e' | \epsilon \\
t \rightarrow num t' \\
t' \rightarrow '/ ' num t' | \epsilon
\]

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

\* Don’t need to consider associativity and precedence for HW4

- but you should understand the issues
- and how to resolve them
The Visitor Design Pattern

The Visitor pattern allows:

1. functions over an object structure (like an AST) to be decoupled from the object structure itself
2. this means you can have many different functions, without having to change the object structure

- an object (node) in the structure “accepts” a visitor
- which means the node simply passes itself to the visitor
- the visitor then “visits” the node (e.g., prints, evaluates, etc.)
- and then “navigates” to child nodes (repeating the process)
A simple/hypothetical example

In PrintVisitor:

```java
public class PrintVisitor {

    public void visit(ValueNode v) {
        system.out.println(v.value);
    }

    public void visit(PlusNode v) {
        v.leftExpr.accept(this);
        system.out.println(" + ");
        v.rightExpr.accept(this);
    }

    public void visit(TimesNode v) {
        v.leftExpr.accept(this);
        system.out.println(" * ");
        v.rightExpr.accept(this);
    }
}
```
Implementing Visitors in Python ...

- The previous examples relied on function overloading
- But a given function can have at most one definition in Python
- Plus no explicit types

The general solution is to define the classes as follows ...

class PrintVisitor(object):
    def __init__(self):
        pass
    def visit_value_node(self, value_node):
        print(value_node.value)
    def visit_plus_node(self, plus_node):
        plus_node.leftExpr.accept(self)
        print('+')
        plus_node.rightExpr.accept(self)
    def visit_times_node(self, plus_node):
        plus_node.leftExpr.accept(self)
        print('*')
        plus_node.rightExpr.accept(self)

class ExprNode(object): pass

class ValueNode(ExprNode):
    def __init__(self):
        self.value = None
    def accept(visitor):
        visitor.visit_value_node(self)
class PlusNode(ExprNode):
    def __init__(self):
        self.leftExpr = None
        self.rightExpr = None
    def accept(visitor):
        visitor.visit_plus_node(self)

class TimesNode(ExprNode):
    def __init__(self):
        self.leftExpr = None
        self.rightExpr = None
    def accept(visitor):
        visitor.visit_times_node(self)

How we'll use the Visitor Pattern:

- In HW4 we use the visitor pattern to print nodes
- In HW5 we'll use it to do type checking of an AST
- In HW6 we'll use it to interpret (“run”) an AST

All without having to modify the AST classes!