Lecture 13:

- Abstract Syntax Trees

Announcements:

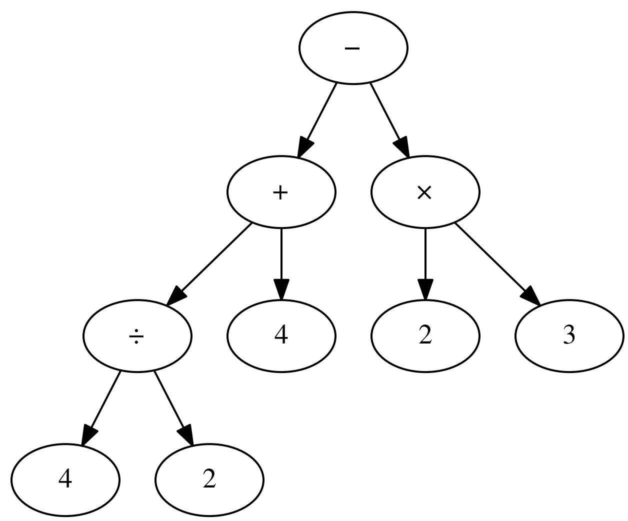
- HW-2 out
- *Note:* Exam next week (Wed), no class Mon

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" ...



   – do "**in-order traversal**" (left, node, right) to "execute" expression tree

   – more node types in an AST, e.g., declarations, loops, var asignment, etc.

Running Example: *with <expr> resurrected*

<stmt_list> ::= <stmt> <stmt_list_tail>

<stmt_list_tail> ::= SEMICOLON <stmt_list> | ϵ

<stmt> ::= VAR ASSIGN <expr>

<expr> ::= VAR <expr_tail>

<expr_tail> ::= PLUS VAR | MINUS VAR | ϵ

The AST might contain nodes (objects) representing:

- statement lists (`StmtList`)
- an assignment with a var and an expression (`Stmt`)
- expressions with single var and (optional) op and expression (`Expr`)

We'll be using "*Plain-Old Data*" (POD) classes ... "*Data Classes*" in Python

```
@dataclass
class Expr:
    lhs: Token
    op: Token
    rhs: Token


@dataclass
class Stmt:
    var: Token
    expr: Expr


@dataclass
class StmtList:
    stmts: List[Stmt]
```

```python
def parse(self):
    self.advance()                          # init lexer
    stmt_list_node = StmtList([])           # empty statement list
    self.stmt_list(stmt_list_node)          # descend into stmt_list
    eat(TokenType.EOS, "...")               # ensure EOS
    return stmt_list_node                   # return AST root node


def stmt_list(self, stmt_list_node):
    stmt_node = Stmt(None, None)            # empty Stmt
    self.stmt(stmt_node)                    # descend into stmt
    stmt_list_node.smts.append(stmt_node) # add the stmt
    self.stmt_list_tail(stmt_list_node)    # continue to tail


def stmt_list_tail(self, stmt_list_node):
    if self.match(TokenType.SEMICOLON):    # check for semicolon
        self.advance()                      # advance past it
        self.stmt_list(stmt_list_node)     # descend into stmt list


def stmt(self, stmt_node):
    stmt_node.var = self.curr_token         # store var token
    self.eat(TokenType.VAR, "...")          # ensure VAR
    self.eat(TokenType.ASSIGN, "...")       # ensure ASSIGN
    expr_node = Expr(None, None, None)     # empty expr node
    self.expr(expr_node)                    # descend into expr
    stmt_node.expr = expr_node              # connect expr node
```

Check In: Rewrite the remaining recursive descent functions to build the AST