Lecture 14:

- Exam 1 overview
- AST exercise
- AST navigation (intro)

Announcements:

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

Exam 1 Overview

Basics:

- Closed notes
- Worth 50 points
- 4 multipart questions

Possible Topics: *Everything fair game since start of semester*

- Compiler and interpreter components (e.g., draw the pipeline)
- Differences between compilers and interpreters
- Lexical analysis (tokens, token streams, how lexers work)
- Grammars (derivations, $LL(k)$, writing grammars)
- Recursive descent parsing (given grammar, write functions)

To study:

- Quiz yourself from the lecture notes
- Redo lecture note check-ins
- Re-answer the quizzes from scratch
- Re-answer the exercise on recursive descent

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
```

Example AST Classes:

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


@dataclass
class Stmt:
    var: Token
    expr: Expr


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

```
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

For example:

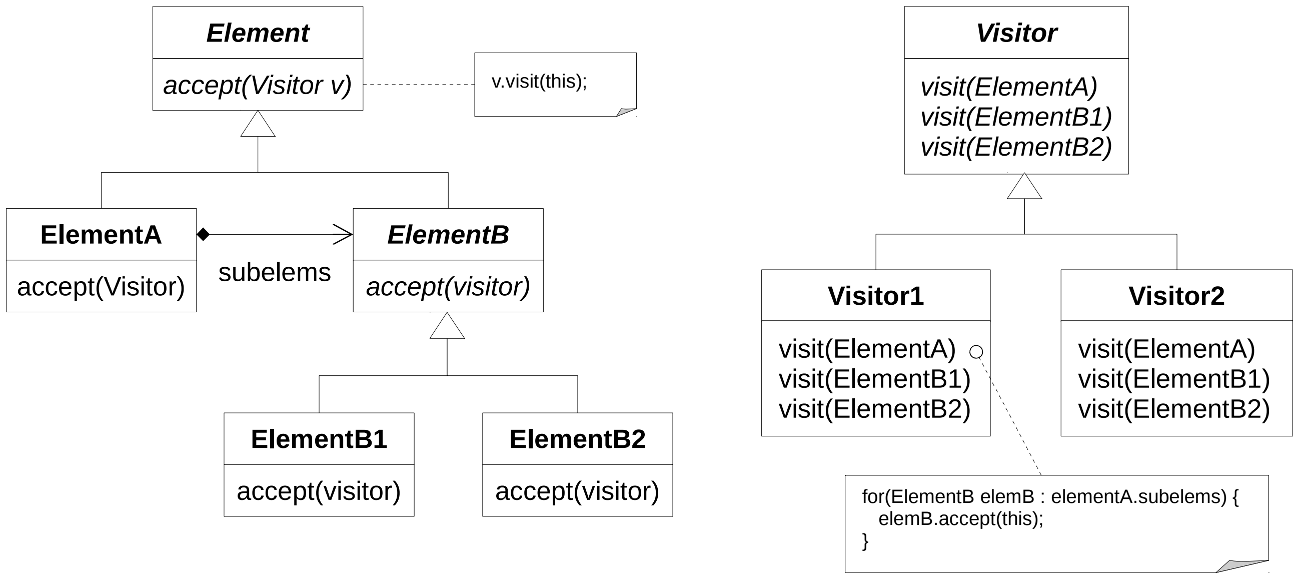
```
def expr(self, expr_node):
    expr_node.lhs = self.curr_token         # store var token
    self.eat(TokenType.VAR, "...")          # ensure VAR
    self.expr_tail(expr_node)               # descend into tail


def expr_tail(self, expr_node):
    ops = [TokenType.PLUS, TokenType.MINUS]
    if self.match_any(ops):                 # check for op
        expr_node.op = self.curr_token      # store the op
        self.advance()                      # move past op
        expr_node.rhs = self.curr_token     # store the var
        self.eat(TokenType.VAR, '...')      # ensure VAR
```

Check In: Draw the AST (object graph) resulting from "`A = B + C; B = A`"

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)

For those interested in why visitor developed, see "**double dispatch**" ...

A Simple Example in Python

- *Note*: Python doesn't support function overloading – have to deviate a bit

The Visitor: *Have to encode data type in function name*

```
class Visitor:

    def visit_expr(self, expr):
        pass

    def visit_stmt(self, stmt):
        pass

    def vist_stmt_list(self, stmt_list):
        pass
```

The AST Classes: *Add the accept functions*

```
@dataclass
class Expr:
    # ... from before ...
    def accept(self, visitor):
        visitor.visit_expr(self)

@dataclass
class Stmt:
    # ... from before ...
    def accept(self, visitor):
        visitor.visit_stmt(self)

@dataclass
class StmtList:
    # ... from before ...
    def accept(self, visitor):
        visitor.vist_stmt_list(self)
```

A simple Python-based **`PrintVisitor`**:

```python
class PrintVisitor(Visitor):

    # helper function
    def output(msg):
        print(msg, end='')

    def visit_expr(self, expr):
        self.output(expr.lhs.lexeme)
        if expr.op is not None:
            self.output(' ' + expr.op.lexeme + ' ')
            self.output(expr.rhs.lexeme)

    def visit_stmt(self, stmt):
        self.output(expr.var.lexeme)
        self.output(' = ')
        stmt.expr.accept(self)         # have expr print itself

    def visit_stmt_list(self, stmt_list):
        for i in range(len(stmt_list.stmts)):
            stmt_list.stmts[i].accept(self)
            # check that isn't last stmt
            if i < len(stmt_list.stmts) - 1:
                self.output(';\n')
        # newline at end
        self.output('\n')
```