Lecture 12:

- Quiz 3
- Abstract Syntax Trees (ASTs)

Announcements:

- HW-2 due Wed

From Last Time: AST Generation

Another Example:

<stmt_list> ::= (<stmt>)*

<stmt> ::= <assign_stmt> | <while_stmt>

<assign_stmt> ::= ID ASSIGN <expr> SEMICOLON

<while_stmt> ::= WHILE <expr> LBRACE <stmt_list> RBRACE

<expr> ::= (ID | INT_VAL) (PLUS <expr> | LESS <expr> | ε)

```
class StmtList {
  List<Stmt> stmts = new ArrayList<>();
}

class Stmt {}  // ... or interface

class AssignStmt extends Stmt {
  Token varName;
  Expr expr;
}
```

```
class WhileStmt extends Stmt {
  Expr condition;
  StmtList stmts;
}

class Expr {
  Token lhs;
  Optional<Token> op = Optional.emtpy();
  Optional<Expr> rhs = Optional.empty();
}
```

From Last Time: AST Generation

The recursive descent functions to build AST:

```
public StmtList parse() {
  advance();
  StmtList rootNode = stmtList();
  eat(TokenType.EOS, "...");
  return rootNode;
}

public StmtList stmtList() {
  StmtList stmtListNode = new StmtList();
  while (matchAny(List.of(TokenType.ID, TokenType.WHILE))) {
    advance();
    stmtListNode.stmts.add(stmt());
  }
  return stmtListNode;
}

public Stmt stmt() {
  if (match(TokenType.ID))
    return assignStmt();
  else
    return whileStmt();
}
```


From Last Time: AST Generation

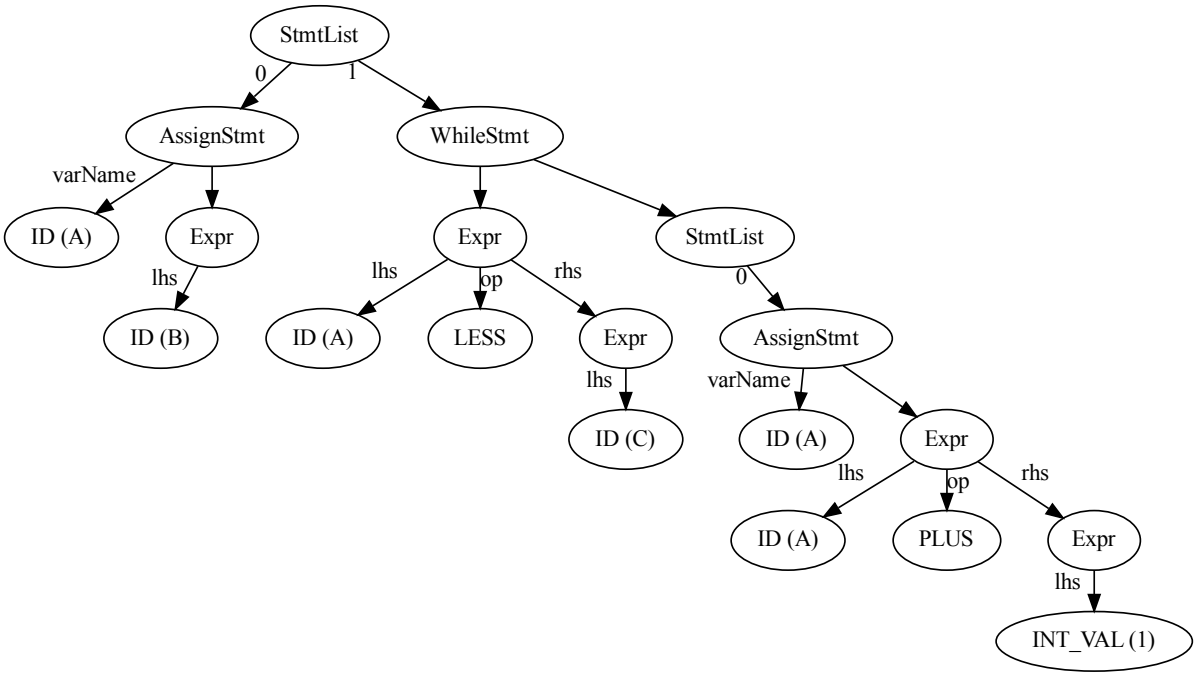
```
public AssignStmt assignStmt() {
  AssignStmt stmtNode = new AssignStmt();
  stmtNode.varName = currToken;
  eat(TokenType.ID, "...");
  eat(TokenType.ASSIGN, "...");
  stmtNode.expr = expr();
  return stmtNode;
}

public WhileStmt whileStmt() {
  WhileStmt stmtNode = new WhileStmt();
  eat(TokenType.WHILE, "...");
  stmtNode.condition = expr();
  eat(TokenType.LBRACE, "...");
  stmtNode.stmts = stmtList();
  eat(TokenType.RBRACE, "...");
}
```

From Last Time: AST Generation

```
public Expr expr() {
  Expr exprNode = new Expr();
  exprNode.lhs = currToken;
  eat(TokenType.ID, "...");
  if (matchAny(List.of(TokenType.PLUS, TokenType.LESS))) {
    exprNode.op = Optional.of(currToken);
    exprNode.rhs = Optional.of(expr());
  }
  return exprNode;
}
```


From Last Time: AST Generation

The AST (object graph) for "`A = B; while A < C { A = A + 1 }`"



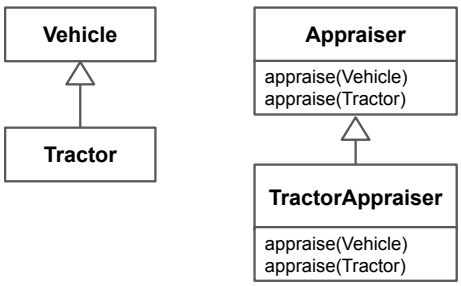
Double Dispatch Problem: Static vs Dynamic Decisions

Check in: What does the following print?

```
class Vehicle {}
class Tractor extends Vehicle {}

class Appraiser {
  void appraise(Vehicle v) {System.out.println("Appraising vehicle");}
  void appraise(Tractor t) {System.out.println("Appraising tractor");}
}
class TractorAppraiser extends Appraiser {
  void appraise(Vehicle v) {System.out.println("Boring");}
  void appraise(Tractor t) {System.out.println("I LOVE TRACTORS");}
}

class Main {
  public static void main(String[] args) {
    Appraiser a1 = new Appraiser();
    Vehicle v = new Tractor();
    a1.appraise(v);                          // Appraising vehicle
    Appraiser a2 = new TractorAppraiser();
    a2.appraise(v);                          // Boring
    Tractor t = new Tractor();
    a2.appraise(t);                          // I LOVE TRACTORS
  }
}
```


Double Dispatch Problem: Static vs Dynamic Decisions

Example double-dispatch clases in pictures:



In **single dispatch** ... `obj.f(v)`

- select where to find `f` based on `obj`'s dynamic (runtime) type
- select version of `f` based on `v`'s static (declared) type

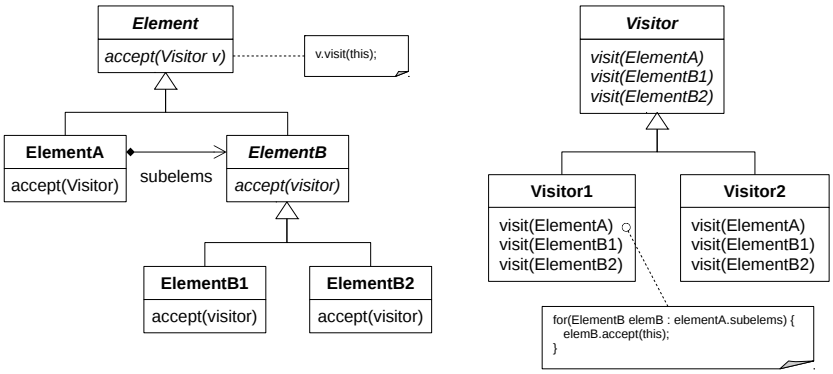
In the example, however, we want **dynamic dispatch** ...

- select both based on the dynamic (runtime) types

The Visitor Design Pattern

The **Visitor Pattern**:

1. decouple functions on object structure (e.g., AST) from structure itself
2. allows different functions, without changing object structure



- objects (nodes) ***accept*** a visitor
- accepting turns around and calls `visit` ... to avoid double dispatch!
- `visit` functions process nodes and navigate (via `accept` calls)