Lecture 13:

- Exam 1 Overview
- Abstract Syntax Trees (ASTs)
- MyPL AST Overview

Announcements:

- HW-3 out

Exam 1 Overview

Basics:

- Closed notes, etc.
- Worth 60 points (10% of final grade)
- Four multipart questions

Notes:

- Everything is fair game, no questions on AST creation and traversal
- There will be coding questions (in Java)
- Compilers, interpreters, transpilers, MyPL
- Lexical analysis / lexer
- Context free grammars and $LL(k)$
- Recursive descent parsing
- Use lecture notes, extended notes, check ins, quizzes, homeworks

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;
}
```

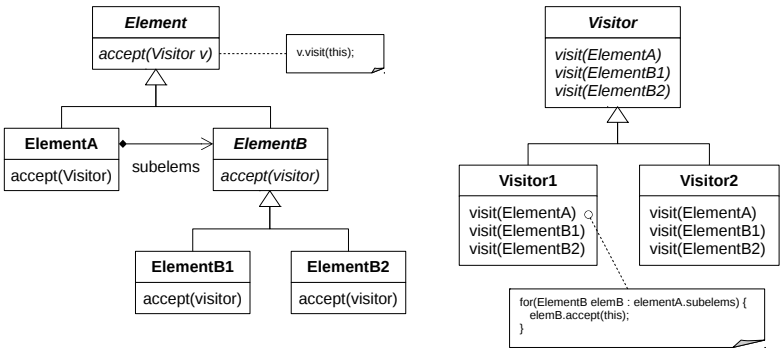
```
class WhileStmt extends Stmt {
  Expr condition;
  StmtList stmts;
}

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


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)

Visitor Design Pattern Example

The **Visitor** interface:

```
public interface Visitor {
  public void visit(StmtList stmtListNode);
  public void visit(AssignStmt stmtNode);
  public void visit(WhileStmt stmtNode);
  public void visit(Expr exprNode);
}
```

Extending the AST classes to support the Visitor:

```
public class StmtList {
  ... same as before ...
  public void visit(Visitor v) {v.accept(this);}
}
public class AssignStmt { ...
  public void visit(Visitor v) {v.accept(this);}
}
public class WhileStmt { ...
  public void visit(Visitor v) {v.accept(this);}
}
public class Expr { ...
  public void visit(Visitor v) {v.accept(this);}
}
```


Visitor Design Pattern Example

Implementing a **pretty print visitor**:

```
public class PrintVisitor implements Visitor {
  private void write(String s) {System.out.print(s);}  // helper function

  public void visit(StmtList stmtListNode) {
    for (Stmt stmtNode : stmtListNode.stmts)
      stmtNode.accept(this);
  }

  public void visit(AssignStmt stmtNode) {
    write(stmtNode.varName.lexeme);
    write(" = ");
    stmtNode.expr.accept(this);
    write(";\n");
  }

  public void visit(WhileStmt stmtNode) {
    write("while ");
    stmtNode.condition.accept(this);
    write("{\n");
    stmtNode.stmts.accept(this);   // need to deal with indentation! (HW-3)
    write("}\n");
  }
```

Visitor Design Pattern Example

```
... continued ...


public void visit(Expr exprNode) {
    write(exprNode.lhs.lexeme);
    if (exprNode.op.isPresent()) {
      write(" " + exprNode.op.get().lexeme + " ");
      exprNode.rhs.get().accept(this);
    }
  }
}
```