Tree Walking

- All except for the simplest of language applications build ASTs as their IRs
- These ASTs need to be processed ➔ Tree Walking
- Read Chaps 6&7 (reference)
Tree Walking

- Tree walking is usually done in “depth first” manner
- Operators and operands can be processed in different orders:
  - Pre-order: root node first then children, e.g. (+ (* 3 2) 4)
  - In-order: nodes are processed in lexical order, e.g. (3 * 2) + 4
  - Post-order: children nodes are processed first and then the root node, e.g. ((3 2 *) 4 +)
Tree Walking

- We have seen tree walking already in two previous applications
  - Bytecode interpreter – here our tree was just a linked list of instruction nodes – a unary tree
  - Pretty printer with a twist – here our tree was a true AST
// abstract base class for our bytecode instruction set
public abstract class Instr {
    private Instr nextInstr = null;
    
    public Instr() {
        nextInstr = null;
    }
    
    public Instr(Instr i) {
        nextInstr = i;
    }

    public void putNext(Instr i) {
        nextInstr = i;
    }

    public abstract void executeInstr();
    public final void executeNextInstr() {
        if (nextInstr != null) {
            nextInstr.executeInstr();
        } else {
            System.out.println("Execution all done.");
        }
    }
}

// implementation of the store instruction
import java.util.*;

public class StoreInstr extends Instr {
    private String varName;
    private Expr exp;
    private HashMap memory;
    public StoreInstr(String v, Expr e, HashMap m) {
        varName = v;
        exp = e;
        memory = m;
    }

    public void executeInstr() {
        // implement the behavior
        int expVal = exp.eval();
        memory.put(varName, new Integer(expVal));
        // transfer control to the next instruction
        executeNextInstr();
    }
}

Tree Walker
public abstract class AST {
    private ArrayList<AST> children = new ArrayList<AST>();

    public final void addAST(AST child) {
        children.add(child);
    }

    public final AST getAST(int ix) {
        return children.get(ix);
    }

    public final void printIndent(int r) {
        for (int i = 0; i < r; i++)
            System.out.print("     ");
    }

    public int size() {
        return children.size();
    }

    public abstract void pprint(int indent);
}

public class WhileStmt extends Stmt {
    public WhileStmt(Expr e, Stmt s) {
        this.addAST(e);
        this.addAST(s);
    }

    public void pprint(int ix) {
        printIndent(ix);
        System.out.print("while ( ");
        // pretty print the expression
        this.getAST(0).pprint(ix);
        System.out.println(")");
        // pretty print the while body
        this.getAST(1).pprint(ix+1);
    }
}
Embedded Tree Walking

- The tree walkers we have looked at so far are called *embedded tree walkers*
  - **Pros:**
    - Easy to implement
  - **Cons:**
    - Mixing program representation with the behavior of specific language processing phases,
    - Tree walker is distributed over many different classes - difficult to maintain (each node class has its own tree walking code)
Tree Walking Visitor

- Separate tree walking code from AST
  - Pros:
    - very clean code,
    - tree walker resides centrally in one class separate from the AST representation,
    - easy to add additional visitors to a single representation without changing the representation
  - Cons:
    - separate representation and tree walking classes need to stay in sync.
Tree Walking Visitor

Input → build → AST → write → Output

Visitors (tree processing)
Tree Walking Visitor Architecture

Dispatcher

AST

Visitor Code

Contains an action for each AST node type.

Selects the correct action to dispatch depending on the AST node type.
Pretty Printer with Visitor

- Let’s rewrite our Pretty Printer with a Twist
- We move the pretty printing code from the AST to a separate pretty printing visitor
IR Design

Note: the structure of the AST has not changed we simply moved the pretty printing behavior to the visitor class.
IR Design with Use Cases
Abstract Base Class for AST

```java
import java.util.*;

public abstract class AST {
    private ArrayList<AST> children = new ArrayList<AST>();
    public final void addAST(AST child) {
        children.add(child);
    }
    public final AST getAST(int ix) {
        return children.get(ix);
    }
    public final int size() {
        return children.size();
    }
}
```

Assignment Statement

```java
public class AssignStmt extends Stmt {
    private String lhsvar;
    public AssignStmt(String v, Expr e) {
        lhsvar = v;
        this.addAST(e);
    }
    public String lhsVar() {
        return lhsvar;
    }
}
```

While Statement

```java
public class WhileStmt extends Stmt {
    public WhileStmt(Expr e, Stmt s) {
        this.addAST(e);
        this.addAST(s);
    }
}
```

No pretty printing code!
public class PrettyPrintVisitor {

    // public dispatcher for the pretty print visitor - we take advantage of the
    // reflective nature of Java and ask for the class name of the current AST object
    // and use the class name to dispatch the correct pretty print behavior.
    public void dispatch(AST ast, int ix) {
        if (ast.getClass() == AssignStmt.class) pprint((AssignStmt)ast,ix);
        else if (ast.getClass() == BlockStmt.class) pprint((BlockStmt)ast,ix);
        else if (ast.getClass() == GetStmt.class) pprint((GetStmt)ast,ix);
        else if (ast.getClass() == IfStmt.class) pprint((IfStmt)ast,ix);
        else if (ast.getClass() == PutStmt.class) pprint((PutStmt)ast,ix);
        else if (ast.getClass() == WhileStmt.class) pprint((WhileStmt)ast,ix);
        else if (ast.getClass() == StmtList.class) pprint((StmtList)ast,ix);
        else if (ast.getClass() == BinopExpr.class) pprint((BinopExpr)ast,ix);
        else if (ast.getClass() == NumExpr.class) pprint((NumExpr)ast,ix);
        else if (ast.getClass() == ParenExpr.class) pprint((ParenExpr)ast,ix);
        else if (ast.getClass() == VarExpr.class) pprint((VarExpr)ast,ix);
        else {
            System.out.println("PrettyPrintVisitor: unknown class type");
            System.exit(1);
        }
    }

    //****** pretty print statement level ASTs
    // pretty print statements
    private void pprint(AssignStmt ast, int ix) {
        printIndent(ix);
        System.out.print(ast.lhsVar + " = ");
        this.dispatch(ast.getAST(0),ix);
        // if not in usage map then defined but not used ,
        // flag it
        if (simple1Parser.usageMap.get(ast.lhsVar) == null) {
            System.out.print("// -- var "+ast.lhsVar+" unused -- ");
        }
        System.out.println();
    }

    // pretty print while statements
    private void pprint(WhileStmt ast, int ix) {
        printIndent(ix);
        System.out.print("while ( ");
        // pretty print the expression
        this.dispatch(ast.getAST(0),ix);
        System.out.println(")");
        // pretty print the while body
        this.dispatch(ast.getAST(1),ix+1);
    }

    ...
}

Note: Here we use a simplified version of the visitor, true visitors
written in OO style use a double dispatch. For more info see the
book *Design Patterns: Elements of Reusable Object-Oriented
Software* by Erich Gamma, Richard Helm, Ralph Johnson and
John Vlissides.
import org.antlr.runtime.*;
import java.util.*;

public class PPrint {
    public static void main(String[] args) throws Exception {

        // check if we have an input file, if not print out error and abort
        if (args.length == 0) {
            System.out.println("Usage: java PPrint <input file>");
            System.exit(0);
        } else {
            System.out.println("Processing: " + args[0]);
        }

        // set up and initialize our lexer and parser objects
        // open up the input file
        ANTLRFileStream input = new ANTLRFileStream(args[0]);
        // create the lexer with the input stream
        simple1Lexer lexer = new simple1Lexer(input);
        // create a token stream for the parser
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        // create a parser object with the token stream
        simple1Parser parser = new simple1Parser(tokens);

        // call the toplevel recursive descent parsing function to construct
        // our IR
        AST ast = parser.prog();

        // pretty print our AST
        PrettyPrintVisitor visitor = new PrettyPrintVisitor();
        visitor.dispatch(ast,0);
    }
}
Example Code

- Code is available as Simple1PPVisitor on the website.
Assignment

- Assignment #5 – see webpage.