Type Systems and the Visitor Design Pattern

```java
public class WhileLoopCounter extends DefaultASTVisitor {
    private int numWhileLoops = 0;

    @Override
    public void preVisit(ASTWhileLoop node) {
        numWhileLoops++;
    }

    @Override
    public void postVisit(ASTProgram node) {
        System.out.println("Number of while loops = " + numWhileLoops);
    }
}
```
General theme

- **Pattern matching** over a tree is very useful in compilers
  - Debug output (P3)
  - **Type checking & other static analysis (P4)**
  - Code generation (P5)
  - Instruction selection
- Theory and practice
  - **Type systems** describe correctly-typed program trees
  - **Visitor design pattern** allows clean implementation in a non-functional language
    - Generalization of **tree traversal** (CS 240 approach)
Types

- A type is an abstract category characterizing a range of data values
  - Base types: integer, character, boolean, floating-point
  - Enumerated types (finite list of constants)
  - Pointer types (“address of X”)
  - Array or list types (“list of X”)
  - Compound/record types (named collections of other types)
  - Function types: (type1, type2, type3) → type4
Type Systems

- A type system is a set of type rules
  - Rules: valid types, type compatibility, and how values can be used
  - A type judgment is an assertion that expression $x$ has type $t$
    - Often requires the context of a type environment (i.e., symbol table)
  - “Strongly typed” if every expression can be assigned an unambiguous type
  - “Statically typed” if all types can be assigned at compile time
  - “Dynamically typed” if some types can only be discovered at runtime

- Benefits of a robust type system
  - Earlier error detection
  - Better documentation
  - Increased modularization
A formal type system is a set of type inference rules

- Each rule has a name, zero or more premises (above the line), and a conclusion (below the line)
- Premises and conclusions are type judgments (A ⊢ x : t)
- “⊢” is a ternary operator connecting expressions with types
- Omit type for statements (“A ⊢ s” means “s is well-typed in environment A”)
Formal Type Theory

- **Type proofs** consist of composing multiple type rules
  - Apply rule instances recursively to form proof trees
  - **Type environments** (e.g., symbol tables, marked in rules with \( \vdash \) operator) provide type context information
  - Proof structure is based on the AST structure ("syntax-directed")
  - **Curry-Howard correspondence** ("proofs as programs")

\[
\begin{align*}
A & = \{ \text{foo : int} \rightarrow \text{int}, x : \text{int}, y : \text{int} \} \\
TAssign & \quad \text{foo(y) + 1} : \vdash \text{int} \\
TDec & \quad A \vdash 1 : \text{int} \\
TAdd & \quad A \vdash x = \text{foo(y)} + 1 \\
TFuncCall & \quad \text{foo : (int) \rightarrow int} \in A \\
TVar & \quad y : \text{int} \in A \\
TDec & \quad x : \text{int} \in A \\
\end{align*}
\]
Formal Type Theory

• Is the following Decaf expression well-typed in the given environment?
  – If so, what is its type?

\[
x + 4
\]

\[
A = \{ \ x : \text{int} \ \}
\]

AST:

```
BinExpr (+)
```

```
Loc (x)    Lit (4)
```
Formal Type Theory

\[
\begin{align*}
\text{TLoc} & \quad \frac{\text{ID} : \tau \in \Gamma}{\Gamma \vdash \text{ID} : \tau} \\
\text{TDec} & \quad \vdash \text{DEC} : \text{int} \\
\text{TAdd} & \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 \, +' \, e_2 : \text{int}} \\
\end{align*}
\]

\[
\begin{align*}
\text{TLoc} & \quad \frac{x : \text{int} \in A}{A \vdash x : \text{int}} \\
\text{TDec} & \quad \frac{A \vdash 4 : \text{int}}{A \vdash x + 4 : \text{int}} \\
\end{align*}
\]

A = \{ x : \text{int} \}
Formal Type Theory

\[ \begin{align*}
\text{TLoc} & : \frac{\text{ID} : \tau \in \Gamma}{\Gamma \vdash \text{ID} : \tau} \\
\text{TDec} & : \frac{}{\vdash \text{DEC} : \text{int}} \\
\text{TAdd} & : \frac{\Gamma \vdash e_1 : \text{int}, \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}}
\end{align*} \]
P4: Static Analysis

- Language and project specifications provide rules to check at each type of AST node while traversing the AST
  - E.g., at ASTWhileLoop, make sure the conditional has a boolean type
  - E.g., at ASTBinaryExpr, if it’s an add make sure both operands are integers (or if it’s an equals make sure the operand types match)

\[
\begin{align*}
TDec & \quad \vdash DEC : \text{int} \\
THex & \quad \vdash \text{HEX} : \text{int} \\
TStr & \quad \vdash \text{STR} : \text{str} \\
TTrue & \quad \vdash \text{true} : \text{bool} \\
TFalse & \quad \vdash \text{false} : \text{bool} \\
TSubExpr & \quad \frac{\Gamma \vdash e : t}{\Gamma \vdash (\text{‘‘} e \text{‘‘}) : t} \\
TAdd & \quad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \\
TEq & \quad \frac{\Gamma \vdash e_1 : t \quad \Gamma \vdash e_2 : t}{\Gamma \vdash e_1 = e_2 : \text{bool}} \\
TWhile & \quad \frac{\Gamma \vdash e : \text{bool} \quad \Gamma \vdash b}{\Gamma \vdash \text{while ‘‘} e \text{‘‘} b}
\end{align*}
\]
• General idea: traverse AST and reject invalid programs
  - Need to traverse the tree multiple times
    • Build symbol tables
    • Perform type checking
    • Later compiler passes
  - We could write the tree traversal code every time, but that would get tedious and would result in a lot of code duplication
    • Software engineering provides a better way in the form of the visitor design pattern
A brief digression ...

• What are "design patterns"?

(HINT: remember them from CS 345!)
A brief digression ...

• What are "design patterns"?
  – A reusable "template" or "pattern" that solves a common design problem
    • "Tried and true" solutions
  – Main reference: Design Patterns: Elements of Reusable Object-Oriented Software
    • "Gang of Four:" Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
Common Design Patterns

- **Adapter** – Converts one interface into another
- **Factory** – Allows clients to create objects without specifying a concrete class
- **Flyweight** – Manages large numbers of similar objects efficiently via sharing
- **Iterator** – Provides sequential access to a collection
- **Monitor** – Ensures mutually-exclusive access to member variables
- **Null Object** – Prevents null pointer dereferences by providing "default" object
- **Observer** – Track and update multiple dependents automatically on events
- **Singleton** – Provides global access to a single instance object
- **Strategy** – Encapsulate interchangeable algorithms
- **Thread Pool** – Manages allocation of available resources to queued tasks
- **Visitor** – Provides an iterator over a (usually recursive) structure
Design Patterns

• Pros
  – Faster development
  – More robust code (if implemented properly)
  – More readable code (for those familiar with the patterns)
  – Improved maintainability

• Cons
  – Increased abstraction
  – Increased complexity
  – Philosophical: Suggests language deficiencies
    • Solution: Consider using a different language
Visitor Pattern

- Visitor design pattern: don't mix data and actions
  - Separates the **representation** of an object structure from the definition of **operations** on that structure
  - Keeps data class definitions cleaner
  - Allows the creation of new operations without modifying all data classes
  - Solves a general issue with OO languages
    - Lack of **multiple dispatch** (choosing a concrete method based on two objects' data types)
      - NOTE: This is stronger than parametric polymorphism alone
    - Less useful in functional languages with more robust pattern matching
General Form

- Data: Abstract **Element** (ASTNode)
  - ConcreteElement1 (ASTProgram)
  - ConcreteElement2 (ASTVariable)
  - ConcreteElement3 (ASTFunction)
  - etc.
  - All elements define "Accept()" method that recursively calls "Accept()" on any child nodes (this is the actual tree traversal code!)

- Actions: Abstract **Visitor** (DefaultASTVisitor)
  - ConcreteVisitor1 (BuildParentLinks)
  - ConcreteVisitor2 (CalculateNodeDepths)
  - ConcreteVisitor3 (StaticAnalysis)
    - BuildSymbolTables
    - MyDecafAnalysis
  - All visitors have "VisitX()" methods for each element type
Benefits

• Adding new operations is easy
  – Just create a new concrete visitor
  – In our compiler, create a new DefaultASTVisitor subclass

• No wasted space for state in data classes
  – Just maintain state in the visitors
  – In our compiler, we will make a few exceptions for state that is shared across many visitors (e.g., symbol tables)
Drawbacks

• Adding new data classes is hard
  – This won't matter for us, because our AST types are dictated by the grammar and won't change

• Breaks encapsulation for data members
  – Visitors often need access to all data members
  – This is ok for us, because our data objects are basically just structs anyway (all data is public)
Minor Modifications

- "Accept()" → "traverse()"
- "Visit()" → "preVisit()" and "postVisit()"
  - preVisit() allows preorder operations
  - postVisit() allows postorder operations
  - Also, a single inorder method: inVisit(ASTBinaryExpr)
- DefaultASTVisitor class
  - Implements ASTVisitor interface
  - Contains empty implementations of all "visit" methods
  - Allows subclasses to define only the relevant visit methods
Visitor example

```java
public class WhileLoopCounter extends DefaultASTVisitor {
    private int numWhileLoops = 0;

    @Override
    public void preVisit(ASTWhileLoop node) {
        numWhileLoops++;
    }

    @Override
    public void postVisit(ASTProgram node) {
        System.out.println("Number of while loops = " + numWhileLoops);
    }
}

In DecafCompiler.java:
ast.traverse(new WhileLoopCounter());
```
Decaf Project

• Project 3
  - ASTVisitor
  - DefaultASTVisitor (implements ASTVisitor)
    - PrintDebugTree
    - ExportTreeDOT
    - BuildParentLinks (activity)
    - CalculateNodeDepths (activity)

• Project 4
  - PrintDebugSymbolTables (extends DefaultASTVisitor)
  - StaticAnalysis (extends DefaultASTVisitor)
    - BuildSymbolTables
    - DecafAnalysis + MyDecafAnalysis

• Project 5
  - ILOCGenerator + MyILOCGenerator