Type Systems and the Visitor Design Pattern
• *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

Not all of these will be necessary for Decaf
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 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 ($A \vdash x : t$)
- “$\vdash$” is a ternary operator connecting type environments, expressions, and types
- Omit type for statements (“$A \vdash s$” means “$s$ is well-typed in environment $A$”)

\[
\begin{align*}
\text{TDec} & : \Gamma \vdash \text{DEC} : \text{int} \\
\text{TTrue} & : \Gamma \vdash \text{true} : \text{bool} \\
\text{TLoc} & : \Gamma \vdash \text{ID} : \tau \in \Gamma \\
\text{TAdd} & : \Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int} \\
& \quad \Gamma \vdash e_1 + e_2 : \text{int} \\
\text{TAAssign} & : \Gamma \vdash \text{ID} : \tau \in \Gamma \\
& \quad \Gamma \vdash \text{ID} \ ' = ' e \ ' ; ' \\
\text{TFuncCall} & : \text{ID} : (\tau_1, \tau_2, \ldots, \tau_n) \rightarrow \tau_r \in \Gamma \\
& \quad \Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2 \quad \ldots \quad \Gamma \vdash e_n : \tau_n \\
& \quad \Gamma \vdash \text{ID} \ ' ( ' e_1, e_2, \ldots, e_n \ ' ) ' : \tau_r
\end{align*}
\]
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) provide type context
  - Proof structure is based on the AST structure (“**syntax-directed**”)
  - **Curry-Howard correspondence** (“proofs as programs”)

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

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

\[
\text{TLoc} \quad \frac{x : \text{int} \in A}{A \vdash x : \text{int}} \quad \frac{A \vdash 4 : \text{int}}{A \vdash x + 4 : \text{int}}
\]

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

\[
\begin{align*}
T_{Dec} & \vdash DEC : \text{int} \\
T_{Hex} & \vdash HEX : \text{int} \\
T_{Str} & \vdash STR : \text{str} \\
T_{True} & \vdash \text{true : bool} \\
T_{False} & \vdash \text{false : bool} \\
T_{SubExpr} & \frac{\Gamma \vdash e : t}{\Gamma \vdash (\text{‘} e \text{‘}) : t} \\
T_{Add} & \frac{\Gamma \vdash e_1 : \text{int}}{\Gamma \vdash e_1 \text{‘}+\text{‘} e_2 : \text{int}} \quad (\text{similar for } T_{Sub} (-), T_{Mul} (*), T_{Div} (/) \text{ and } T_{Mod} (\%) ) \\
T_{Eq} & \frac{\Gamma \vdash e_1 : t}{\Gamma \vdash e_1 \text{‘}==\text{‘} e_2 : \text{bool}} \quad (\text{similar for } T_{Neq} (\text{‘}!\text{‘}) ) \\
T_{While} & \frac{\Gamma \vdash e : \text{bool}}{\Gamma \vdash \text{while ‘(‘} e \text{‘)‘} \text{‘} b} 
\end{align*}
\]
P4: Static Analysis

• 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 be tedious w/ 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"?

(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
- **ThreadPool** – 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 most OO languages
    - Lack of *multiple dispatch* (choosing a concrete method based on two objects' data types)
      - NOTE: This is stronger than single dispatch + overloading alone
    - Less useful in functional languages with more robust pattern matching
class A {
    public void foo(A a) { System.out.println("A::foo(A)"); }
    public void foo(B b) { System.out.println("A::foo(B)"); }
}

class B extends A {
    public void foo(A a) { System.out.println("B::foo(A)"); }
    public void foo(B b) { System.out.println("B::foo(B)"); }
}

A a1 = new A();
B b1 = new B();
A a2 = b1;

<table>
<thead>
<tr>
<th></th>
<th>SINGLE DISPATCH</th>
<th>MULTIPLE DISPATCH</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1.foo(a1);</td>
<td>A::foo(A)</td>
<td>A::foo(A)</td>
</tr>
<tr>
<td>b1.foo(a1);</td>
<td>B::foo(A)</td>
<td>B::foo(A)</td>
</tr>
<tr>
<td>a1.foo(b1);</td>
<td>A::foo(B)</td>
<td>A::foo(B)</td>
</tr>
<tr>
<td>b1.foo(b1);</td>
<td>B::foo(B)</td>
<td>B::foo(B)</td>
</tr>
<tr>
<td>a1.foo(a2);</td>
<td>A::foo(A)</td>
<td>A::foo(B)</td>
</tr>
<tr>
<td>b1.foo(a2);</td>
<td>B::foo(A)</td>
<td>B::foo(B)</td>
</tr>
<tr>
<td>a2.foo(a2);</td>
<td>B::foo(A)</td>
<td>B::foo(B)</td>
</tr>
</tbody>
</table>
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
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**