CSE 401 – Compilers

ASTs, Modularity, and the Visitor Pattern
Hal Perkins
Winter 2017
Agenda

• Today:
  – AST operations: modularity and encapsulation
  – Visitor pattern: basic ideas and variations
  – Some of the “why” behind the “how”

• Covered in sections:
  – Representation of ASTs as a tree of Java objects
  – Parser semantic actions and AST generation
  – AST/Parser/Visitor classes in project code
Abstract Syntax Trees (ASTs - review)

• Idea: capture the essential structure of a program; omit extraneous details
  – i.e., only what the rest of the compiler needs; omit things used only to guide the parse (e.g., punctuation, chain productions)

• Java implementation
  – Simple tree node objects (basically structs/records)
    • In addition to subtree pointers, usually include other useful information like source program locations (e.g., line/character numbers), links to semantic (symbol table) information (later), ...
    • But not much more!
  – Use type system and inheritance to factor common information and allow polymorphic treatment of related nodes
Operations on ASTs

• Once we have the AST, we may want to:
  – Print a readable dump of the tree (pretty printing)
  – Do static semantic analysis:
    • Type checking
    • Verify that things are declared and initialized properly
    • Etc. etc. etc. etc.
  – Perform optimizing transformations on the tree
  – Generate code from the tree, or
  – Generate another IR from the tree for further processing
Modularity

• Classic slogans:
  – Do one thing well
  – Minimize coupling, maximize cohesion
  – Isolate operations/abstractions in modules
  – Hide implementation details

• Okay, so where does the typechecker module in MiniJava belong?
Where do the Operations Go?

• Pure “object-oriented” style
  – Really, really, really smart AST nodes
  – Each node knows how to perform every operation on itself

```java
public class WhileNode extends StmtNode {
    public WhileNode(...);
    public typeCheck(...);
    public StrengthReductionOptimize(...);
    public generateCode(...);
    public prettyPrint(...);
    ...
}
```
Critique

• This is nicely encapsulated – all details about a WhileNode are hidden in that class
• But it is poor modularity
• What happens if we want to add a new Optimize (or any other) operation?
  – Have to open up every node class
• Furthermore, it means that the details of any particular operation (optimization, type checking) are scattered across the node classes
Modularity Issues

• Smart nodes make sense if the set of operations is relatively fixed, but we expect to need flexibility to add new kinds of nodes

• Example: graphics system
  – Operations: draw, move, iconify, highlight
  – Objects: textbox, scrollbar, canvas, menu, dialog box, plus new objects defined as the system evolves
Modularity in a Compiler

• Abstract syntax does not change frequently over time
  \[ \therefore \text{Kinds of nodes are relatively fixed} \]
• As a compiler evolves, it is common to modify or add operations on the AST nodes
  – Want to modularize each operation (type check, optimize, code gen) so its parts are together
  – Want to avoid having to change node classes when we modify or add an operation on the tree
# Two Views of Modularity

<table>
<thead>
<tr>
<th></th>
<th>draw</th>
<th>move</th>
<th>iconify</th>
<th>highlight</th>
<th>transmogrify</th>
</tr>
</thead>
<tbody>
<tr>
<td>circle</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>text</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>canvas</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>scroll</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>dialog</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Type check</th>
<th>Optimize</th>
<th>Generate x86</th>
<th>Flatten</th>
<th>Print</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDENT</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>exp</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>while</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>if</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>Binop</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Visitor Pattern

- Idea: Package each operation (optimization, print, code gen, ...) in a separate visitor class
- Create exactly one instance of each visitor class (singleton pattern)
  - Sometimes called a “function object”
  - Contains all of the methods for that particular operation, one for each kind of AST node
- Include a generic “accept visitor” method in every node class
- To perform an operation, pass the appropriate “visitor object” around the AST during a traversal
Avoiding instanceof

- We’d like to avoid huge if-elseifif nests in the visitor to discover the node types

```java
void checkTypes(ASTNode p) {
    if (p instanceof WhileNode) { ... }
    else if (p instanceof IfNode) { ... }
    else if (p instanceof BinExp) { ... }
    ...
}
```
Visitor Double Dispatch

• Include a “visit” method for every AST node type in each Visitor
  
  void visit(WhileNode);
  void visit(ExpNode);
  etc.

• Include an accept(Visitor v) method in each AST node class

• When Visitor v is passed to an AST node, the node’s accept method calls v.visit(this)
  – Selects correct Visitor method for this node
  – “Double dispatch”
Visitor Interface

interface Visitor {
    // overload visit for each AST node type
    public void visit(WhileNode s);
    public void visit(IfNode s);
    public void visit(BinExp e);
    ...
}

– Every separate Visitor implements this interface
– Aside: The result type can be whatever is convenient, doesn’t have to be void, although that is common
Accept Method in Each AST Node Class

• Every AST class overrides accept(Visitor)
• Example
  
  ```java
  public class WhileNode extends StmtNode {
      ...
      // accept a visit from a Visitor object v
      public void accept(Visitor v) {
          v.visit(this); // dynamic dispatch on “this” (WhileNode)
      }
      ...
  }
  
  • Key points
    – Visitor object passed as a parameter to WhileNode
    – WhileNode calls visit, which dispatches to visit(WhileNode) automatically – i.e., the correct method for this kind of node
Composite Objects (1)

• How do we handle composite objects?
• One possibility: the accept method passes the visitor down to subtrees before (or after) visiting itself

```java
public class WhileNode extends StmtNode {
    Expr exp; Stmt stmt; // children
    ...
    // accept a visit from visitor v
    public void accept (Visitor v) {
        this.exp.accept(v);
        this.stmt.accept(v);
        v.visit(this);
    }
}```
Composite Objects (2)

• Another possibility: the visitor can control the traversal

```java
public void visit(WhileNode p) {
    p.expr.accept(this);
    p.stmt.accept(this);
}
```
Encapsulation

• A visitor object often needs to be able to access state in the AST nodes
  ∴ May need to expose more node state than we might do otherwise
  • i.e., lots of public fields in node objects
    – Overall a good tradeoff – better modularity
      (plus, the nodes are relatively simple data objects anyway – not hiding much of anything)
Visitor Actions and State

• A visitor function has a reference to the node it is visiting (the parameter)
  \[\therefore\] can access and manipulate subtrees directly
• Visitor object can also include local data (state) shared by methods in the visitor
  – This data is effectively “global” to the methods that make up the visitor object, and can be used to store and pass around information

```java
public class TypeCheckVisitor extends NodeVisitor {
  public void visit(WhileNode s) { ... }
  public void visit(IfNode s) { ... }
  ...
  private <local state>; // all methods can read/write this
}
```
So which to choose?

• Possibilities:
  – Node objects drive the traversal and pass the visitors around the tree in standard ways
  – Visitor object drives the traversal (the visitor has access to the node, including references to child subtrees)

• In a compiler:
  – First choice handles many common cases
  – Big compilers often have multiple visitor schemes (e.g., several different traversals defined in Node interface + custom traversals in some visitors)
  – For MiniJava: keep it simple and start with supplied examples, but if you really need to do something different, you can
    • (i.e., keep an open mind, but not so open that you create needless complexity)
Why is it so complicated?

• What we’re really trying to do: 2-argument dynamic dispatch
  – Pick correct method to execute based on dynamic types of both the node and the visitor

• But Java and most O-O languages only support single dispatch
  – So we use it twice to get the effect we want
References

• For Visitor pattern (and many others)
  – *Design Patterns: Elements of Reusable Object-Oriented Software*, Gamma, Helm, Johnson, and Vlissides, Addison-Wesley, 1995 (the classic, examples are in old C++ and Smalltalk)

• Specific information for MiniJava AST and visitors in Appel textbook & online
Coming Attractions

• Static Analysis
  – Type checking & representation of types
  – Non-context-free rules (variables and types must be declared, etc.)

• Symbol Tables

• & more

• But first, we should finish parsing – LL(1) & recursive descent

• Later, more about compiler IRs when we get to optimizations