CSE 401/M501 – Compilers

ASTs, Modularity, and the Visitor Pattern

Hal Perkins

Autumn 2018

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 concrete syntax used only to guide the parse (e.g., punctuation, chain productions)
- Java implementation
  - Simple tree node objects (basically structs/records)
    - Subtree pointers plus (usually) 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
  - Print a parseable (source-code) version 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

```
public class WhileNode extends StmtNode {
  public WhileNode(…);
  public typeCheck(…);
  public StrengthReductionOptimize(…);
  public DeadCodeEliminationOptimize(…);
  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 ☹
- Worse: 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 and 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 during execution or over lifetime of system
- Another example: objects in a game or simulation

Modularity in a Compiler

- Abstract syntax does not change frequently over time – language changes are usually incremental
  ∴ 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 located together in the source code
  – Want to avoid having to change node classes when we modify or add an operation on the tree

Two Views of Modularity

	draw	move	iconify	highlight	transmogrify
circle	X	X	X	X	X
text	X	X	X	X	X
canvas	X	X	X	X	X
scroll	X	X	X	X	X
dialog	X	X	X	X	X
…					

	Type check	Optimize	Generate x86	Flatten	Print
IDENT	X	X	X	X	X
exp	X	X	X	X	X
while	X	X	X	X	X
if	X	X	X	X	X
Binop	X	X	X	X	X
…					

Visitor Pattern

- Idea: Package each operation (optimization, print, code gen, …) in a separate visitor class (module)
- Create exactly one instance of each visitor class (a singleton)
  - 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-elseif nests in the visitor to discover the node types

```
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
  - Often called "Double dispatch" (but really single dispatch + overloading)

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
- Note: could also give methods unique names e.g., visitWhile, visitIf, visitBinExp, etc. instead of overloading visit(…). Best to follow existing code, otherwise individual preference.

Accept Method in Each AST Node Class

- Every AST class overrides accept(Visitor)
- Example

```
public class WhileNode extends StmtNode {
    …
    // accept a visit from a Visitor object v
    public void accept(Visitor v) {
      v.visit(this);      // call using type of "this" (WhileNode)
    }                     // and dynamic dispatch to current visitor
    …
}
```

- Key points
  - Visitor object passed as a parameter to WhileNode
  - WhileNode calls visit, which calls visit(WhileNode) because of compile-time overloading – i.e., the correct method for this kind of node
- Note: if visitor methods have unique names instead of overloading visit(…) then WhileNode would call something like v.visitWhile(this).

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

```
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 inside the visit method for that particular kind of node

```
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 to 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)
  ∴ can access and manipulate subtrees directly
- Visitor object can also contain local data (state) shared by methods in the visitor
  - This data is effectively "global" to the methods in the visitor object, and can be used to store and pass around information accumulated by the visit methods

```
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 standard traversals defined in Node interface plus 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 single dispatch plus overloading 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 C++ and Smalltalk)
  - *Object-Oriented Design & Patterns*, Horstmann, A-W, 2nd ed, 2006 (uses Java)
- 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

- Then compiler back end

- More about compiler IRs when we get to optimizations

- But first: finish parsing (LL, top-down, recursive descent, …)