

CSE 401 - Compilers

Interlude: ASTs, Modularity, and the Visitor Pattern Hal Perkins Winter 2009

© 2002-09 Hal Porkins & HW CSF



Modularity

Classic slogans:





- Do one thing well
- Minimize coupling, maximize cohesion
- Isolate operations/abstractions in modules
- Hide implementation details
- OK, so where's the typechecker module in MiniJava?

© 2002-09 Hal Perkins & LIW CSF



Operations on ASTs

- In a typical compiler, we may want to do these things with the AST:
 - 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 (often flatten to a linear IR)

2/22/2009

© 2002-09 Hal Perkins & UW CSE



Where do the Operations Go?

- Pure "object-oriented" style
 - Smart AST nodes
 - Each node knows how to perform every operation

public class WhileNode extends StmtNode { public typeCheck(...); public generateCode(...): public prettyPrint(...);

Basically the organization in our MiniJava project

2/22/2009 © 2002-09 Hal Perkins & UW CSE



Critique

- This is nicely encapsulated all details about a WhileNode are hidden in that class
- But there are issues with modularity
- What happens if we want to add a new operation?
 - Have to open up every node class
- Furthermore, it means that the details of any particular operation (printing, type checking) are scattered across the node classes

© 2002-09 Hal Perkins & UW CSE



Modularity Issues

- Smart nodes make sense if the set of operations is relatively fixed, particularly if 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

© 2002-09 Hal Perkins & UW CSE

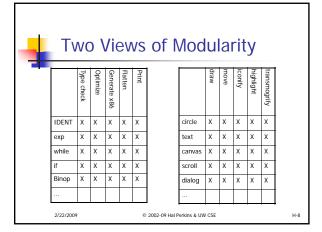
CSE 401 Wi09 H-1



Modularity in a Compiler

- Abstract syntax does not change frequently over time
 - ∴ Kinds of nodes are relatively fixed
- As a compiler evolves, it is more common to modify or add operations
 - Can we modularize each operation (type check, code gen) so its components are together?
 - Can we avoid having to change node classes when we modify or add an operation?

2/22/2009 © 2002-09 Hal Perkins & UW CSE





Visitor Pattern

- Idea: Package each operation in a separate class
 - Contains separate methods for each AST node kind
 - Examples: type check class, flatten class, print class
- Create one instance of this visitor class
 - Sometimes called a "function object"
- Include a generic "accept visitor" method in every node class
- To perform the operation, pass the "visitor object" around the AST during a traversal
 - This object contains separate methods to process each AST node type

2/22/2009 © 2002-09 Hal Perkins & UW CSE



Avoiding instanceof

- Next issue: we'd like to avoid huge if-elseif nests to check the node type in the visitor void checkTypes(ASTNode p) {
 - if (p instanceof WhileNode) { ... }
 else if (p instanceof IfNode) { ... }
 else if (p instanceof BinExp) { ... } ...
- Solution: Include an overloaded "visit" method for each node type and get the node to call back to the correct operation for that node(!)

H-10

H-12

"Double dispatch"

2/22/2009 © 2002-09 Hal Perkins & UW CSE



One More Issue

- We want to be able to add new operations easily, so the nodes shouldn't know anything specific about the actual visitor class(es)
- Solution: an abstract Visitor interface
 - AST nodes include "accept visitor" method for the interface
 - Specific operations (type check, code gen) are implementations of this interface

2/22/2009

© 2002-09 Hal Perkins & UW CSE

Visitor Interface

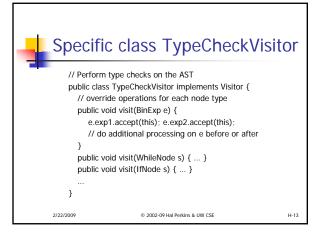
interface Visitor {
 // overload visit for each AST node type
 public void visit(WhileNode s);
 public void visit(IfNode s);
 public void visit(BinExp e);
 ...

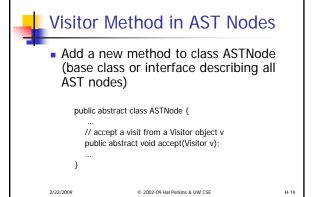
 Aside: The result type can be whatever is convenient, doesn't have to be void

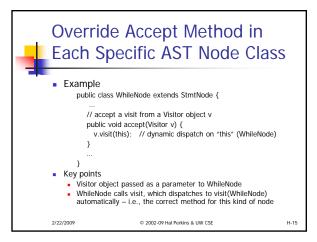
2/22/2009

© 2002-09 Hal Perkins & UW CSE

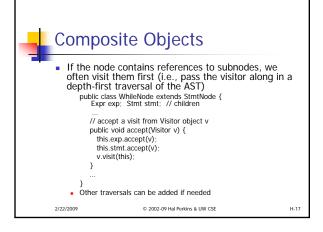
CSE 401 Wi09 H-2

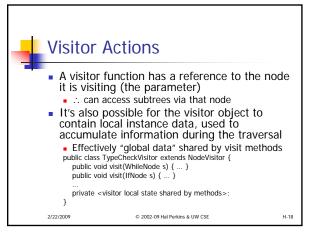












CSE 401 Wi09 H-3



Responsibility for the Traversal

- Possible choices
 - The node objects (as done above)
 - The visitor object (the visitor has access to the node, so it can traverse any substructure it wishes)
 - Some sort of iterator object
- In a compiler, the first choice can handle many common cases

© 2002-09 Hal Perkins & UW CSE



Ouch!

- Does it have to be this complicated?
- What we're trying to do: 2-level dispatch during generic traversal
 - First on the kind of operation (type check, print)
 - Second on the type of the node
- If our language supports double-dispatch we could express this directly
- But in Java and conventional O-O languages, only the first parameter (receiver) controls dispatch
 One solution: multimethods. Research at UW, see papers by Chambers and colleagues

© 2002-09 Hal Perkins & UW CSE



References

- For Visitor pattern (and many others) Design Patterns: Elements of Reusable Object-Oriented Software Gamma, Helm, Johnson, and Vlissides Addison-Wesley, 1995
- Good explanation of how to use visitors in compilers in Appel's Modern Compiler Implementation in Java

2/22/2009

© 2002-09 Hal Perkins & UW CSE

H-21

CSE 401 Wi09 H-4