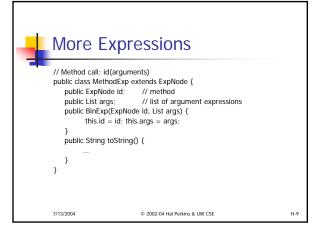


```
More Statement Nodes

// if (exp) stmt [else stmt]
public class IfNode extends StmtNode {
    public ExpNode exp;
    public StmtNode thenStmt, elseStmt;
    public IfNode(ExpNode exp,StmtNode thenStmt,StmtNode elseStmt) {
        this.exp=exp; this.thenStmt=thenStmt;this.elseStmt=elseStmt;
    }
    public IfNode(ExpNode exp, StmtNode thenStmt) {
        this(exp, thenStmt, null);
    }
    public String toString() { ... }
}
```

```
### Company of the Co
```





&c

- These examples are meant to give you some ideas, not necessarily to be used literally
 - E.g., you might find it much better to have a specific AST node for "argument list" that encapsulates the generic java.util.List of arguments
- You'll also need nodes for class and method declarations, parameter lists, and so forth
 - Starter code in book and on web for MiniJava

7/13/2004 © 2002-04 Hal Perkins & UW CSE H-10



Position Information in Nodes

- To produce useful error messages, it's helpful to record the source program location corresponding to a node in that node
 - Most scanner/parser generators have a hook for this, usually storing source position information in tokens
 - Would be nice in our projects, but not required (i.e., get the parser/AST construction working first)

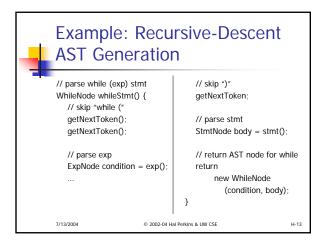
7/13/2004 © 2002-04 Hal Perkins & UW CSE



AST Generation

- Idea: each time the parser recognizes a complete production, it produces as its result an AST node (with, usually, one or more subtrees consisting of the components of the production)
- When we finish parsing, the result of the goal symbol is the complete AST for the program

7/13/2004 © 2002-04 Hal Perkins & UW CSE H-12

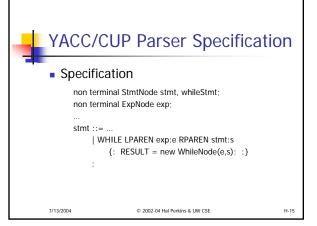




AST Generation in YACC/CUP

- A result type can be specified for each item in the grammar specification
- Each parser rule can be annotated with a semantic action, which is just a piece of Java code that returns a value of the result type
 - The semantic action is executed when the rule is reduced

© 2002-04 Hal Perkins & LIW CSF

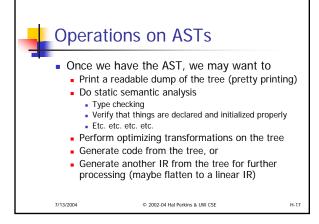




SableCC/JavaCC

- Integrated tools like these provide tools to generate syntax trees automatically
 - Advantage: saves work, don't need to define AST classes and write semantic
 - Disadvantage: generated trees may not be as abstract as we might want

7/13/2004 © 2002-04 Hal Perkins & UW CSE H-16





Where do the Operations Go?

- Pure "object-oriented" style
 - 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 generateCode(...);
 public prettyPrint(...);
```

© 2002-04 Hal Perkins & UW CSE



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 operation?
 - Have to open up every node class
- Furthermore, it means that the details of any particular operation are scattered across the node classes

13/2004

© 2002-04 Hal Perkins & UW CSE

H-21



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

7/13/2004 © 2002-04 Hal Perkins & UW CSE H



Modularity in a Compiler

- Abstract syntax does not change frequently over time
 - .: 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 components are together
 - Want to avoid having to change node classes to modify or add an operation on the tree

7/13/2004

© 2002-04 Hal Perkins & UW CSE



Two Views of Modularity

٠,			_	_	_	_
		Type check	Optimize	Generate x86	Flatten	Print
	IDENT	Х	Х	Х	Х	Х
	ехр	Х	Х	Х	Х	Х
	while	Х	Х	Х	Х	Х
	if	Х	Х	Х	Х	Х
	Binop	Х	Х	Х	Х	Х

	draw	move	iconify	highlight	transmogrify
circle	Х	Х	Х	Х	х
text	Х	Х	Х	Х	х
canvas	Х	Х	Х	Х	х
scroll	Х	Х	Х	Х	х
dialog	Х	Х	Х	Х	х

7/13/2004 © 2002-04 Hal Perkins & UW CSE



Visitor Pattern

- Idea: Package each operation in a separate class
 - One method for each AST node kind
- 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

7/13/2004

© 2002-04 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(!)

 "Double dispatch"

_ _ _ _ _ _ _ _ _ ...

© 2002-04 Hal Perkins & UW CSE

H-24

H-22



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
- 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

7/13/2004 © 2002-04 Hal Perkins & UW CSE

```
visitor Interface

interface Visitor {
    // overload visit for each 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, not necessarily void
```



Specific class TypeCheckVisitor

```
// Perform type checks on the AST
public class TypeCheckVisitor implements Visitor {
    // override operations for each node type
    public void visit(WhileNode s) { ... }
    public void visit(IfNode s) { ... }
    public void visit(BinExp e) {
        e.exp1.accept(this); e.exp2.accept(this);
    }
    ...
}
```

7/13/2004 © 2002-04 Hal Perkins & UW CSE



Add New Visitor Method to AST Nodes

 Add a new method to class ASTNode (base class or interface describing all AST nodes)

```
public abstract class ASTNode {
    ...
    // accept a visit from a Visitor object v
    public abstract void accept(Visitor v);
    ...
}
```

© 2002-04 Hal Perkins & UW CSE H-28



Override Accept Method in Each Specific AST Node Class

Example

public class WhileNode extends StmtNode {
...
// accept a visit from a Visitor object v
public void accept(Visitor v) {

v.visit(this);
}
...

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

7/13/2004

© 2002-04 Hal Perkins & UW CSE



7/13/2004

H-27

Encapsulation

- A visitor object often needs to be able to access state in the AST nodes
 - : May need to expose more state than we might do to otherwise
 - Overall a good tradeoff better modularity
 - (plus, the nodes are relatively simple data objects anyway)

7/13/2004

© 2002-04 Hal Perkins & UW CSE



Composite Objects

 If the node contains references to subnodes, we often visit them first (i.e., pass the visitor along in a depth-first traversal of the AST)

public class WhileNode extends StmtNode {

// accept a visit from Visitor object v
public void accept(Visitor v) {



Visitor Actions

- A visitor function has a reference to the node it is visiting (the parameter)
- It's also possible for the visitor class to contain local instance data, used to accumulate information during the traversal
 - Effectively "global data" shared by visit methods public class TypeCheckVisitor extends NodeVisitor { public void visit(WhileNode s) { ... } public void visit(IfNode s) { ... } ... private <local state>;

7/13/2004 © 2002-04 Hal Perkins & UW CSE



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 will handle many common cases

7/13/2004 © 2002-04 Hall Perkins & UW CSE



H-33

References

- For Visitor pattern (and many others)
 Design Patterns: Elements of Reusable
 Object-Oriented Software
 Gamma, Helm, Johnson, and Vlissides
 Addison-Wesley, 1995
- Specific information for MiniJava AST and visitors in the textbook

7/13/2004 © 2002-04 Hal Perkins & UW CSE H-34



Coming Attractions

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

7/13/2004

© 2002-04 Hal Perkins & UW CSE