


CSE 582 – Compilers

Implementing ASTs (in Java)

Hal Perkins
Autumn 2002


10/24/2002 © 2002 Hal Perkins & UW CSE H-1



Agenda

- n Representing ASTs as Java objects
- n Parser actions
- n Operations on ASTs
 - n Modularity and encapsulation
- n Visitor pattern

10/24/2002 © 2002 Hal Perkins & UW CSE H-2




Review: ASTs

- n An Abstract Syntax Tree captures the essential structure of the program, without the extra concrete grammar details needed to guide the parser
- n AST
- n Example:


```
while ( n > 0 ) {
  n = n - 1;
}
```


10/24/2002 © 2002 Hal Perkins & UW CSE H-3



Representation in Java

- n Basic idea is simple: use small classes as records (or structs) to represent nodes in the AST
 - n Simple data structures, not too smart
- n But also use a bit of inheritance so we can treat related nodes polymorphically

10/24/2002 © 2002 Hal Perkins & UW CSE H-4




Possible AST Nodes for JFlat

```
// Base class of AST node hierarchy
public abstract class ASTNode {
  // operations
  ...
  // string representation
  public String toString() {
    return "somebody didn't override ASTNode.toString()";
  }
  // etc.
}
```

- n Note: In a real compiler, we would usually put the node classes into a separate Java package. Use your own judgment for your project.

10/24/2002 © 2002 Hal Perkins & UW CSE H-5



Some Statement Nodes

```
// Base class for all statements
public abstract class StmtNode extends ASTNode { ... }
// while (exp) stmt
public class WhileNode extends StmtNode {
  public ExpNode exp;
  public StmtNode stmt;
  public WhileNode(ExpNode exp, StmtNode stmt) {
    this.exp = exp; this.stmt = stmt;
  }
  public String toString() {
    return "While(" + exp + ") " + stmt;
  }
}
```

10/24/2002 © 2002 Hal Perkins & UW CSE H-6

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=exp; this.thenStmt=thenStmt; this.elseStmt=null;
    }
    public String toString() { ... }
}
```

10/24/2002

© 2002 Hal Perkins & UW CSE

H-7

Expressions

```
// Base class for all expressions
public abstract class ExpNode extends ASTNode { ... }
// exp1 op exp2
public class BinExp extends ExpNode {
    public ExpNode exp1, exp2; // operands
    public int op; // operator (defined in token class)
    public BinExp(Token op, ExpNode exp1, ExpNode exp2) {
        this.op = op; this.exp1 = exp1; this.exp2 = exp2;
    }
    public String toString() {
        ...
    }
}
```

10/24/2002

© 2002 Hal Perkins & UW CSE

H-8

More Expressions

```
// Method call: id(arguments)
public class MethodExp extends ExpNode {
    public ExpNode id; // method
    public List args; // list of argument expressions
    public MethodExp(ExpNode id, List args) {
        this.id = id; this.args = args;
    }
    public String toString() {
        ...
    }
}
```

10/24/2002

© 2002 Hal Perkins & UW CSE

H-9

&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

10/24/2002

© 2002 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
 - JLex/CUP provides a position field in the Symbols (tokens) to use for this; other parser generators should support something similar
 - Would be nice in our projects, but not required (i.e., get the parser/AST construction working first)

10/24/2002

© 2002 Hal Perkins & UW CSE

H-11

AST Generation

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

10/24/2002

© 2002 Hal Perkins & UW CSE

H-12

Example: Recursive-Descent AST Generation

```
// parse while (exp) stmt      // skip ")"
WhileNode whileStmt() {      getNextToken();
    // skip "while ("
    getNextToken();          // parse stmt
    getNextToken();          StmtNode body = stmt();

    // parse exp              // return AST node for while
    ExpNode condition = exp(); return
    ...                       new WhileNode
                               (condition, body);
}
```

10/24/2002

© 2002 Hal Perkins & UW CSE

H-13

AST Generation in 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

10/24/2002

© 2002 Hal Perkins & UW CSE

H-14

CUP Parser Specification

▪ CUP Specification

```
non terminal StmtNode stmt, whileStmt;
non terminal ExpNode exp;
...
stmt ::= ...
| WHILE LPAREN exp:e RPAREN stmt:s
  { : RESULT = new WhileNode(e,s); :}
;
```

10/24/2002

© 2002 Hal Perkins & UW CSE

H-15

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.
 - Perform optimizing transformations on the tree
 - Generate machine code from the tree, or
 - Generate another IR from the tree for further processing (maybe flatten to a linear IR)

10/24/2002

© 2002 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 generateCode(...);
    public prettyPrint(...);
    ...
}
```

10/24/2002

© 2002 Hal Perkins & UW CSE

H-17

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

10/24/2002

© 2002 Hal Perkins & UW CSE

H-18

Modularity Issues

- Smart nodes make sense if the set of operations is relatively fixed, but we want 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

10/24/2002

© 2002 Hal Perkins & UW CSE

H-19

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.

10/24/2002

© 2002 Hal Perkins & UW CSE

H-20

Two Views of Modularity

	Type check	Optimize	Generate x86	Flatten	Print		draw	move	iconify	highlight	transmogrify
IDENT	X	X	X	X	X						
exp	X	X	X	X	X		X	X	X	X	X
while	X	X	X	X	X		X	X	X	X	X
if	X	X	X	X	X		X	X	X	X	X
Binop	X	X	X	X	X		X	X	X	X	X
...											

10/24/2002

© 2002 Hal Perkins & UW CSE

H-21

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

10/24/2002

© 2002 Hal Perkins & UW CSE

H-22

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: Write one method in the visitor class for each node type and get the node to call back to the correct operation for that node(!)
 - "Double dispatch"

10/24/2002

© 2002 Hal Perkins & UW CSE

H-23

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 NodeVisitor class
 - AST nodes refer to generic class
 - Specific operations (type check, code gen) are realized as subclasses of this class

10/24/2002

© 2002 Hal Perkins & UW CSE

H-24

Abstract class NodeVisitor

```
// Generic NodeVisitor
public abstract class NodeVisitor {
    // declare operations for each node type
    public abstract void visitWhileNode(WhileNode s);
    public abstract void visitIfNode(IfNode s);
    public abstract void visitBinExp(BinExp e);
    ...
}
```

- Aside: The visitor can include methods to visit many different classes, not necessarily related by inheritance

10/24/2002

© 2002 Hal Perkins & UW CSE

H-25

Specific class TypeCheckVisitor

```
// Perform type checks on the AST
public class TypeCheckVisitor extends NodeVisitor {
    // override operations for each node type
    public void visitWhileNode(WhileNode s) { ... }
    public void visitIfNode(IfNode s) { ... }
    public void visitBinExp(BinExp e) { ... }
    ...
}
```

10/24/2002

© 2002 Hal Perkins & UW CSE

H-26

Add New Visitor Method to AST Nodes

- Add a new method to class ASTNode (base class of all AST node classes)

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

10/24/2002

© 2002 Hal Perkins & UW CSE

H-27

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(NodeVisitor v) {
        v.visitWhileNode(this);
    }
    ...
}
```

- Key points

- Visitor object passed as a parameter to WhileNode
- WhileNode calls appropriate method in visitor and passes itself as a parameter, ∴ visitor can access this node!

10/24/2002

© 2002 Hal Perkins & UW CSE

H-28

Encapsulation

- For this to work, the visitor object needs to be able to access any necessary 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)

10/24/2002

© 2002 Hal Perkins & UW CSE

H-29

Composite Objects

- If the node contains references to subnodes, it is common to 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(NodeVisitor v) {
        this.exp.Accept(v);
        this.stmt.Accept(v);
        v.visitWhileNode(this);
    }
    ...
}
```

10/24/2002

© 2002 Hal Perkins & UW CSE

H-30

Visitor Actions

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

10/24/2002

© 2002 Hal Perkins & UW CSE

H-31

## Responsibility for the Traversal

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

10/24/2002

© 2002 Hal Perkins & UW CSE

H-32

## Double Dispatch Revisited

- n Depending on how much faith you have in method overloading, the visitXyzzy methods can all have the same name

```
public class TypeCheckVisitor extends NodeVisitor {
 // override operations for each node type
 public void visit(WhileNode s) { ... }
 public void visit(IfNode s) { ... }
 public void visit(BinExp e) { ... }
 ...
}
```

10/24/2002

© 2002 Hal Perkins & UW CSE

H-33

## Double Dispatch (cont)

- n If we use overloading, the type of the node (this) determines the actual method executed
  - n Something of a question of taste and readability

```
public class WhileNode extends StmtNode {
 ...
 // accept a visit from a Visitor object v
 public void Accept(NodeVisitor v) {
 this.exp.Accept(v);
 this.stmt.Accept(v);
 v.visit(this);
 }
 ...
}
```

10/24/2002

© 2002 Hal Perkins & UW CSE

H-34

## Reference

- n For Visitor pattern (and many others)
  - Design Patterns: Elements of Reusable Object-Oriented Software*
  - Gamma, Helm, Johnson, and Vlissides
  - Addison-Wesley, 1995

10/24/2002

© 2002 Hal Perkins & UW CSE

H-35

## Coming Attractions

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

10/24/2002

© 2002 Hal Perkins & UW CSE

H-36