Implementing ASTs
(in Java)

Hal Perkins
Autumn 2011
Review: ASTs

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

Example:

```plaintext
while ( n > 0 ) {
    n = n - 1;
}
```
Representation in Java

- Basic idea: use small classes as records (structs) to represent AST nodes
  - Simple data structures, not too smart
  - Take advantage of type system
- But also use a bit of inheritance so we can treat related nodes polymorphically
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 (lexical token)
    public BinExp(Token op, ExpNode exp1, ExpNode exp2) {
        this.op = op; this.exp1 = exp1; this.exp2 = exp2;
    }
    public String toString() {
        ...
    }
}
// Method call: id(arguments)
public class MethodExp extends ExpNode {
    public ExpNode id;       // method
    public List args;        // list of argument expressions
    public BinExp(ExpNode id, List args) {
        this.id = id; this.args = args;
    }
    public String toString() {
        ...
    }
}
You’ll also need nodes for class and method declarations, parameter lists, and so forth

For the project we strongly suggest using the AST classes in the starter code, which are taken from the MiniJava website

- Modify if you need to & know what you’re doing
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
  - Included in the MiniJava starter code – good idea to take advantage of it in your code
AST Generation

- Idea: each time the parser recognizes a complete production, it produces as its result an AST node (with links to the subtrees that are the components of the production).
- When we finish parsing, the result of the goal symbol is the complete AST for the program.
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.
YACC/CUP Parser Specification

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

- See the starter code for version with line numbers
Integrated tools like these provide tools to generate syntax trees automatically
- Advantage: saves work; don’t need to define AST classes and write semantic actions
- Disadvantage: generated trees might not have the right level of abstraction for what you want to do

For our project, do-it-yourself with CUP
- Starter code should give the general idea
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
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(...);
    ...
}
```
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
  - ⊴ 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 when we modify or add an operation on the tree
## Two Views of Modularity

<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>

<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>
<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 class
- Create one instance of this **visitor** class
  - 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 the operation, pass the “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

```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
  
  ```java
  void visit(WhileNode);
  void visit(ExpNode);
  etc.
  ```

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

- When Visitor v is passed to AST node, node’s accept method calls v.visit(this)
  
  - Selects correct Visitor method for this node
  - “Double dispatch”
Accept Method in Each AST Node Class

- 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
What if an AST node refers to subnodes?

Visitors often control the traversal

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

Also possible to include more than one kind of accept method in each node to let nodes implement different kinds of traversals

- Probably not needed for MiniJava project
Example TypeCheckVisitor

// Perform type checks on the AST
public class TypeCheckVisitor implements Visitor {
    // override operations for each node type
    public void visit(BinExp e) {
        // visit subexpressions – pass this visitor object
        e.exp1.accept(this); // store its type in var, say, Type type1
        e.exp2.accept(this); // ditto type2
        assert(type1.join(type2).equals(type1)
             || type1.join(type2).equals(type2)); // use a type lattice
    }

    public void visit(WhileNode s) { ... } ...=
}

Encapsulation

- A visitor object often needs to be able to access state in the AST nodes
  - \[\therefore\] May need to expose more node state than we might do to otherwise
- Overall a good tradeoff – better modularity
  - (plus, the nodes are relatively simple data objects anyway – not hiding much of anything)
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, uses C++, Smalltalk)
- Specific information for MiniJava AST and visitors in Appel textbook & online