Agenda

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

- This is a general sketch of the ideas – more details and sample code online for MiniJava
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:

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

- Basic idea is simple: use small classes as records (or structs) to represent nodes in the AST
  - Simple data structures, not too smart
- But also use a bit of inheritance so we can treat related nodes polymorphically
- Following slides sketch the ideas – not necessarily what you’ll use in your project
public abstract class ASTNode {
    // constructors (for convenience)
    ...
    // operations
    ...
    // string representation
    public abstract String toString();
    // visitor methods, etc.
}
Some Statement Nodes

// Base class for all statements
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;
    }
}

(Note on toString: most of the time we’ll want to print the tree in a separate traversal, so this is mostly useful for limited debugging)
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() { ... }
}
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() {
        ...
    }
}
These examples are meant to get across the 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 List of arguments

You’ll also need nodes for class and method declarations, parameter lists, and so forth

- Starter code on the web for MiniJava
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 we distributed – useful 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 in its instance variables)

- When we finish parsing, the result of the goal symbol is the complete AST for the program
Example: Recursive-Descent AST Generation

```java
// parse while (exp) stmt
WhileNode whileStmt() {
    // skip "while ("
    getNextToken();
    getNextToken();
    // parse exp
    ExpNode condition = exp();
    ...

    // skip ")"
    getNextToken;

    // parse stmt
    StmtNode body = stmt();

    // return AST node for while
    return new WhileNode
            (condition, body);
}
```
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
ANTLR/JavaCC/others

- 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
  - The (revised) starter code contains the AST classes from the minijava web site
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 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(...);
    ...
}
```
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 (optimization, type checking) are scattered across the node classes
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>IDENT</th>
<th>Type check</th>
<th>Optimize</th>
<th>Generate x86</th>
<th>Flatten</th>
<th>Print</th>
</tr>
</thead>
<tbody>
<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>
</tbody>
</table>

<table>
<thead>
<tr>
<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>
</tr>
<tr>
<td>text</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>
</tr>
<tr>
<td>scroll</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>
</tr>
</tbody>
</table>

...
Visitor Pattern

- Idea: Package each operation in a separate class
  - One operation 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
  - This object contains separate methods to process each AST node type
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 in each AST node type and get the AST node to call back to the correct operation for that node(!)

  - “Double dispatch”
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
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
Specific class 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); e.exp2.accept(this);
        // do additional processing on e before or after
    }
    public void visit(WhileNode s) { ... }
    public void visit(IfNode s) { ... }
    ...
}

Add Visitor Method to AST Nodes

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

```java
public abstract class ASTNode {
    ...
    // accept a visit from a Visitor object v
    public abstract void accept(Visitor v);
    ...
}
```
Override Accept Method in Each Specific 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
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 – not hiding much of anything)
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) {
          this.exp.accept(v);
          this.stmt.accept(v);
          v.visit(this);
      }
      ...
  }

- Other traversals can be added if needed
Visitor Actions

- A visitor function has a reference to the node it is visiting (the parameter)
  - can access subtrees via that node
- It’s also possible for the visitor object to contain local state (data), used to accumulate information during the traversal
  - Effectively “global data” shared by visit methods

```java
public class TypeCheckVisitor extends NodeVisitor {
    public void visit(WhileNode s) { … }
    public void visit(IfNode s) { … }
    …
    private <local state>;
}
```
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
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 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

- & more