CSE 401 – Compilers

Lecture 10: ASTs, CUP, Visitor Pattern
(a.k.a., everything you need for Project Part II)
Michael Ringenburg
Winter 2013

Agenda

- Representing ASTs as Java objects
- Parser actions: what the parser does when it reduces another step in the derivation.
- Operations on ASTs
  - Modularity and encapsulation
  - Visitor pattern

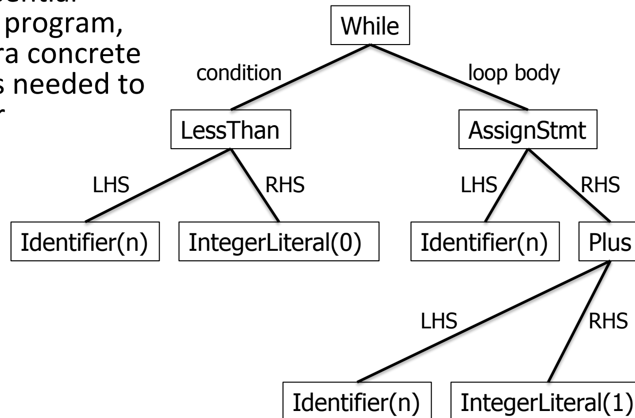
- This is a general sketch of the ideas – more details and examples in the MiniJava web site and project starter code

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:

```
while ( n < 10 ) {
  n = n + 1;
}
```

- AST:



Representation in Java

- Basic idea: use small classes as records (like C structs) to represent AST nodes
  - 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 – do not feel obligated to use literally

Quick Note on 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 Nodes - Sketch

```
// Base class of AST node hierarchy
public abstract class ASTNode {
    // fields that every AST node should share, e.g., line number
    protected int lineNum;
    …
    // constructors (e.g., can set line number, so derived class can do
    // via call to super)
    public int ASTNode(int lnum) { lineNum = lnum; … }
    …
    // Methods every node should implement, e.g., string representation
    // or visitor methods.
    public abstract String toString() ;
    public abstract void accept(Visitor v);
    …
}
```

Some Statement Nodes

```
// Base class for all statements
public abstract class StmtNode extends ASTNode {
    public StmtNode(int ln) { super(ln); … }
    …
}

// while (condition) loopBody
public class WhileNode extends StmtNode {
    public ExpNode condition;
    public StmtNode loopBody;
    public WhileNode(ExpNode cond, StmtNode body, int ln) {
      super(ln);
      this.condition = cond; this.loopBody = body;
    }
    public String toString() {
      return "While(" + condition + ") " + loopBody;
    }
}
```

More Statement Nodes

```
// if (condition) stmt [else stmt]
public class IfNode extends StmtNode {
    public ExpNode condition;
    public StmtNode thenStmt, elseStmt;
    public IfNode(ExpNode cond,StmtNode thenStmt,StmtNode elseStmt, int ln) {
      super(ln);
      this.condition=cond; this.thenStmt=thenStmt;this.elseStmt=elseStmt;
    }
    public IfNode(ExpNode cond, StmtNode thenStmt, int ln) {
      this(cond, thenStmt, null, ln);
    }
    public String toString() { … }
}
```

Expressions

```
// Base class for all expressions
public abstract class ExpNode extends ASTNode {
    public ExpNode(int ln) { super(ln); … }
    …
}

// exp1 op exp2 – alternatively, can have separate class for each operation type, but
// gets unwieldy if you have a lot (not an issue in MiniJava).
public class BinExp extends ExpNode {
    public ExpNode exp1, exp2;     // operands
    public int op;                 // operator (lexical token)
    public BinExp(Token op, ExpNode exp1, ExpNode exp2, int ln) {
      super(ln);
      this.op = op; this.exp1 = exp1; this.exp2 = exp2;
    }
    public String toString() { … }
}
```

More Expressions

```
// Method call: object.method(arguments)
public class MethodExp extends ExpNode {
    public ExpNode object;     // object to invoke method on
    public ExpNode method;     // method to invoke
    public List args;          // list of argument expressions
    public BinExp(ExpNode obj, ExpNode id, List args, int ln) {
      super(ln);
      this.object = obj; this.method = id; this.args = args;
    }
    public String toString() { … }
}
```

More

- These examples are meant to get across the ideas, not necessarily to be used literally
  - E.g., you often see a specific AST node for “argument list” that encapsulates the List of arguments
- You’ll also need nodes for class and method declarations, statement lists, and so forth
- But… 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
  - Let’s take a quick tour … we’ll be looking at src/AST/*.java

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 start symbol production is the complete AST for the program

Example: Top Down Parser AST Generation

```
// 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 production in the grammar specification
  - The type of AST node that should be produced when the parser recognizes that production.
- Each parser rule can be annotated with a semantic action, which is just a piece of Java code
  - Typically this code constructs and returns the appropriate AST node (of result type).
- The semantic action is executed when the rule is reduced

CUP Parser Specification

- Specification

```
nonterminal StmtNode stmt, whileStmt;
nonterminal ExpNode exp;
…
stmt ::= …
     | WHILE:start LPAREN exp:e RPAREN stmt:s
      {: RESULT = new WhileNode(e, s, startleft); :}
     ;
```

- Scanner initializes XXXleft and XXXright variables
  - In theory, XXXleft and XXXright are the positions of the first and last characters of the XXX symbol.
  - MiniJava abuses this slightly. "left" is line number and "right" is column number, both of first character.

Precedence/Associativity

- Older versions of CUP forced you to create unambiguous grammars.
- Newer versions allow you to declare precedence and associativity for operations, and use those to resolve conflicts. making grammar construction simpler:

```
/* Precedence from low to high */
precedence left PLUS, MINUS; // left associativity
precedence left TIMES, DIVIDE;
```

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
  - Starter code should give the general idea
  - Let's take another quick tour, and add Multiplication …

Operations on ASTs

- Once we have the AST, we may want to:
  - Print a readable dump of the tree
  - Do static semantic analysis:
    - Type checking
    - Verify that things are declared and initialized properly
    - 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

```
public class WhileNode extends StmtNode {
  public WhileNode(…);
  public typeCheck(…);
  public optimize1(…);   // Not really – don't typically use ASTs as the IR in
                         //optimization passes
  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 operation, e.g., verifyInitializations()? Or a new optimization pass?
  - 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
  - We'd prefer to have all the code for each analysis/optimization/generation pass together.

Approaches to Modularity

- 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

	Type check	Optimize	Generate x86	Flatten	Print	…
IDENT	X	X	X	X	X	
exp	X	X	X	X	X	
while	X	X	X	X	X	
if	X	X	X	X	X	
Binop	X	X	X	X	X	

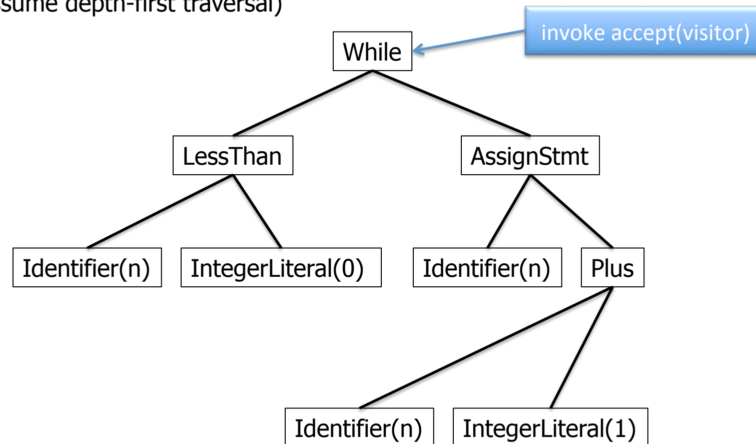
	draw	move	iconify	highlight	transmogrify
circle	X	X	X	X	X
text	X	X	X	X	X
canvas	X	X	X	X	X
scroll	X	X	X	X	X
dialog	X	X	X	X	X
…					

Visitor Pattern

- Idea: Package each operation (optimization, print, code gen, …) in a separate class
  - Contains all of the methods for that particular operation, one for each kind of AST node (via overloaded "visit" method)
- Create one instance of this visitor class
  - Sometimes called a "function object" or "visitor object"
- Include a generic "accept visitor" method in every node class
  - "accept" calls "visit", dynamic dispatch ensures that the "correct" version of "visit" is called
- To perform the operation, pass the "visitor object" around the AST during a traversal

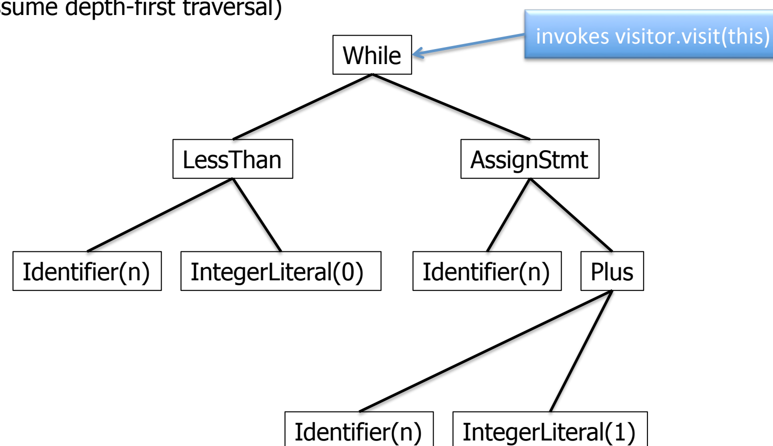
Visitor Pattern, Illustrated

(Assume depth-first traversal)



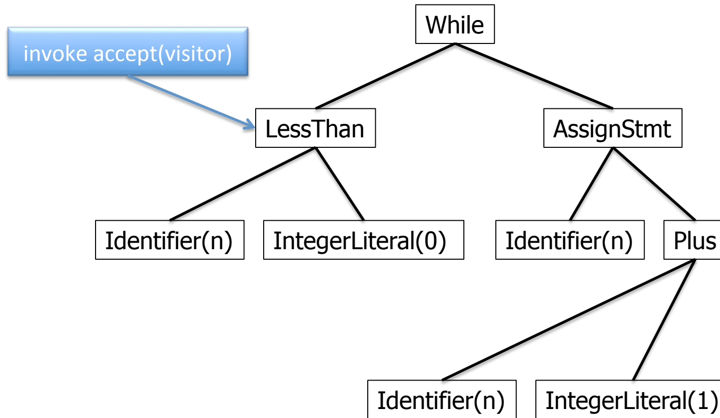
Visitor Pattern, Illustrated

(Assume depth-first traversal)



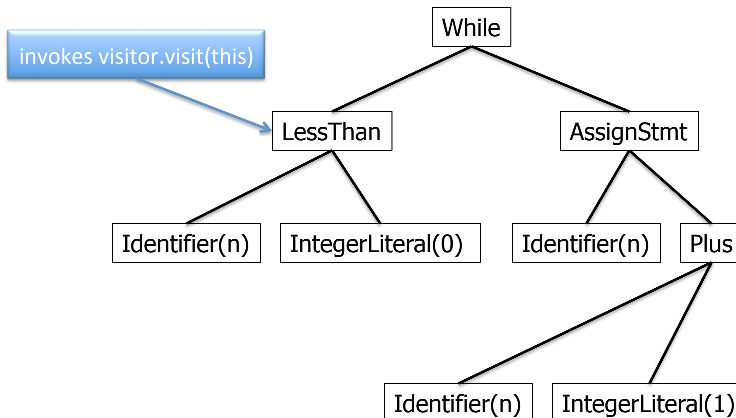
Visitor Pattern, Illustrated

(Assume depth-first traversal)



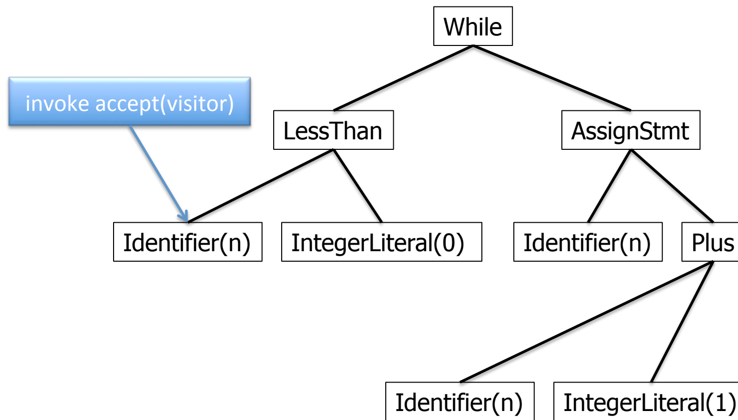
Visitor Pattern, Illustrated

(Assume depth-first traversal)



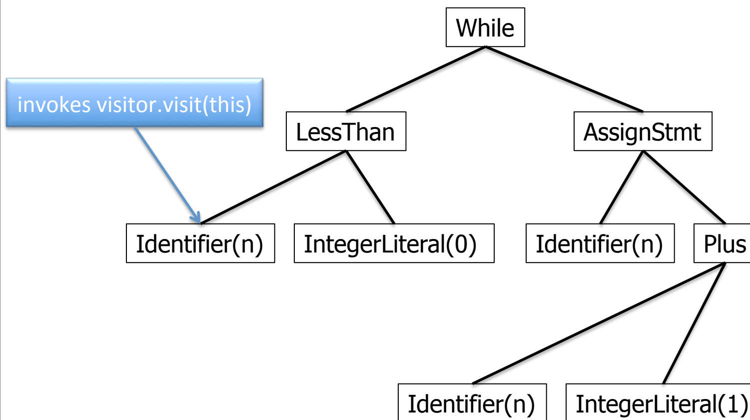
Visitor Pattern, Illustrated

(Assume depth-first traversal)



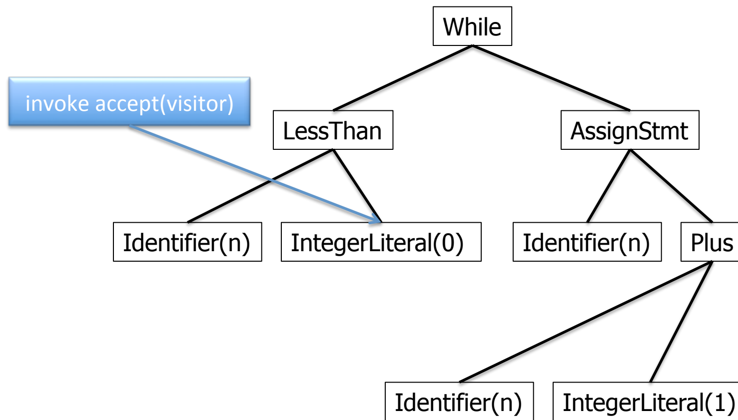
Visitor Pattern, Illustrated

(Assume depth-first traversal)



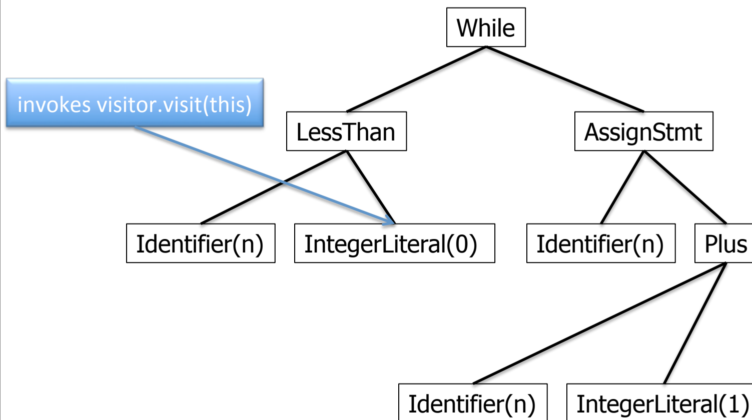
Visitor Pattern, Illustrated

(Assume depth-first traversal)



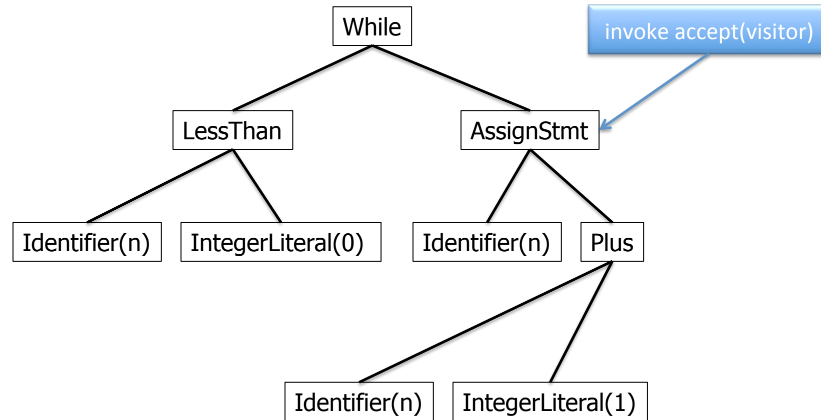
Visitor Pattern, Illustrated

(Assume depth-first traversal)



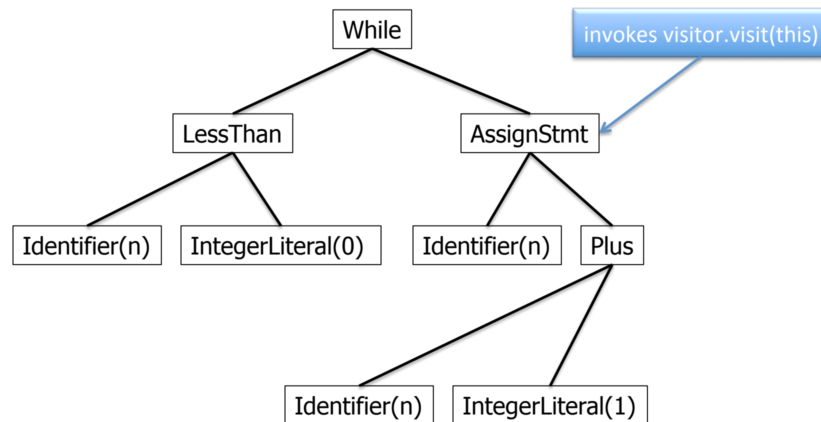
Visitor Pattern, Illustrated

(Assume depth-first traversal)



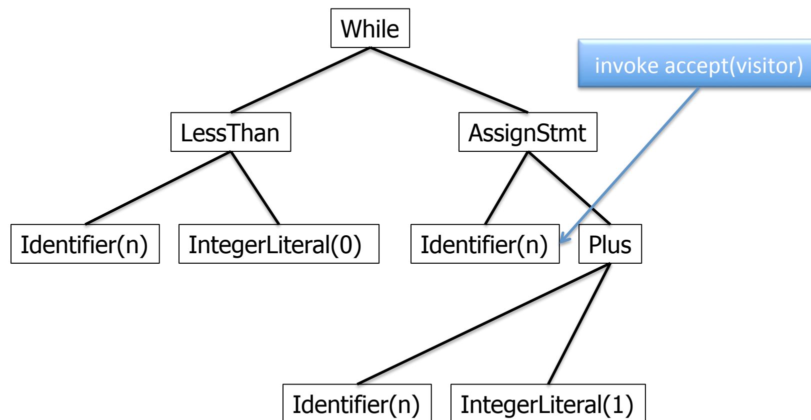
Visitor Pattern, Illustrated

(Assume depth-first traversal)



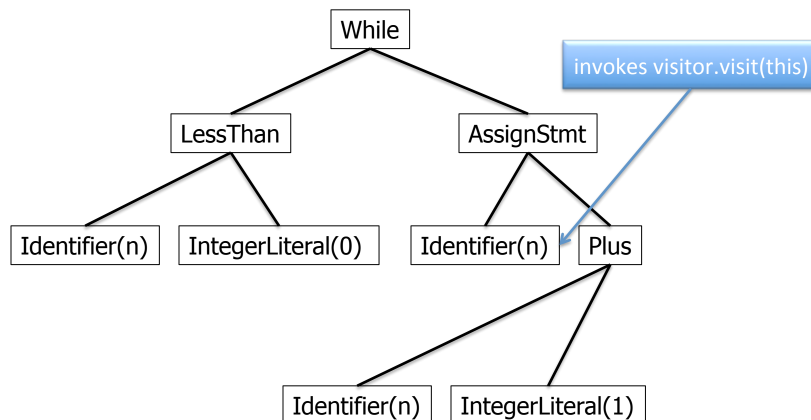
Visitor Pattern, Illustrated

(Assume depth-first traversal)



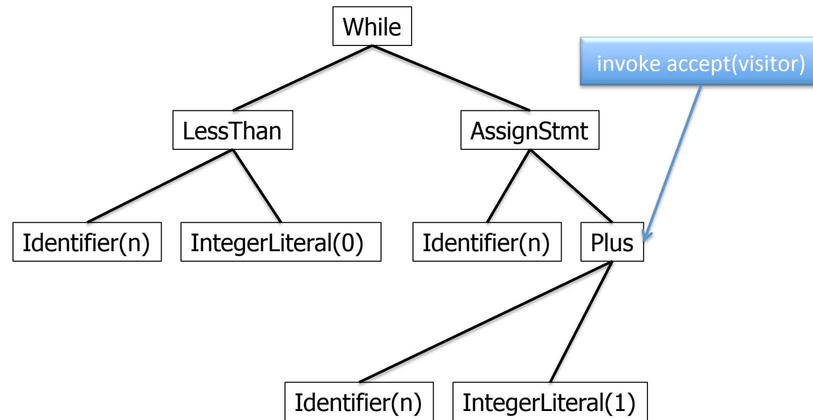
Visitor Pattern, Illustrated

(Assume depth-first traversal)



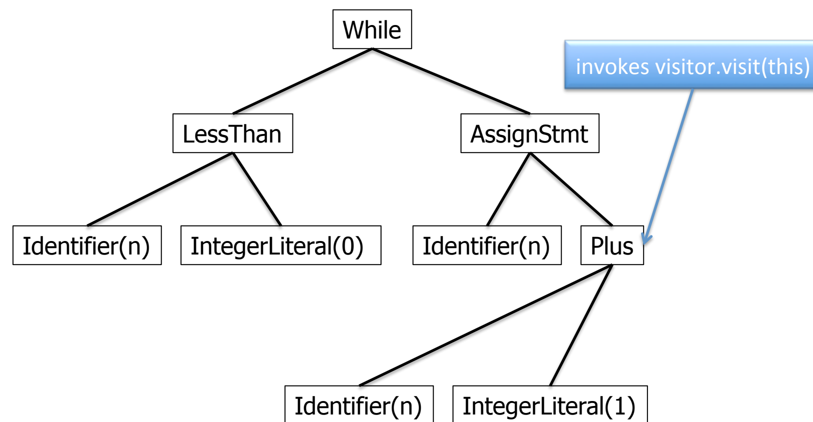
Visitor Pattern, Illustrated

(Assume depth-first traversal)



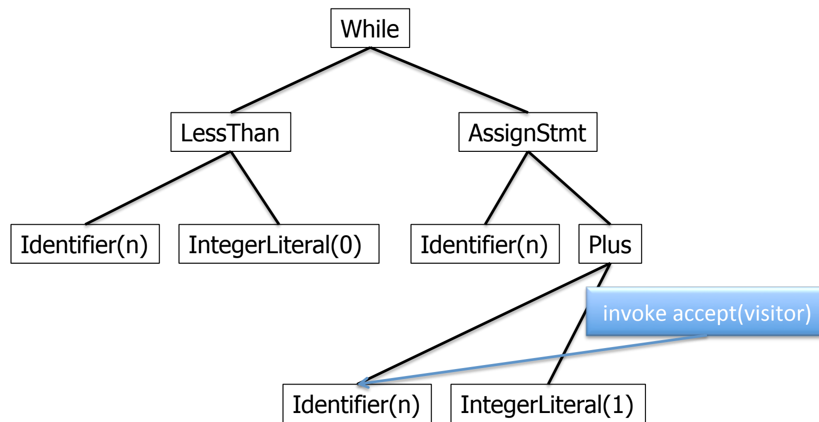
Visitor Pattern, Illustrated

(Assume depth-first traversal)



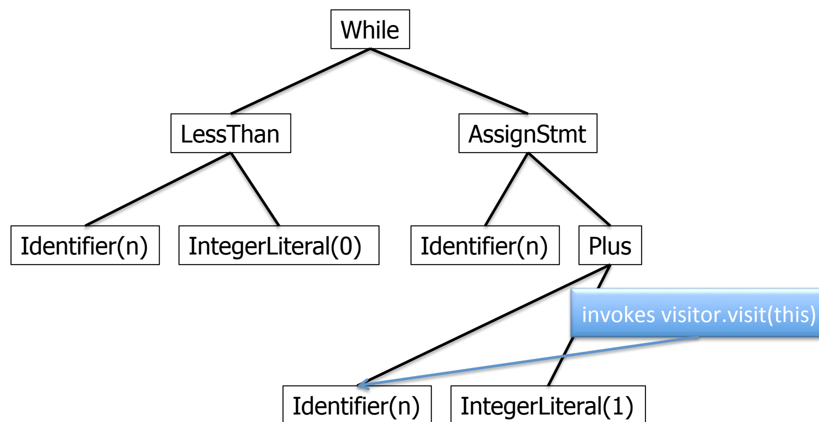
Visitor Pattern, Illustrated

(Assume depth-first traversal)



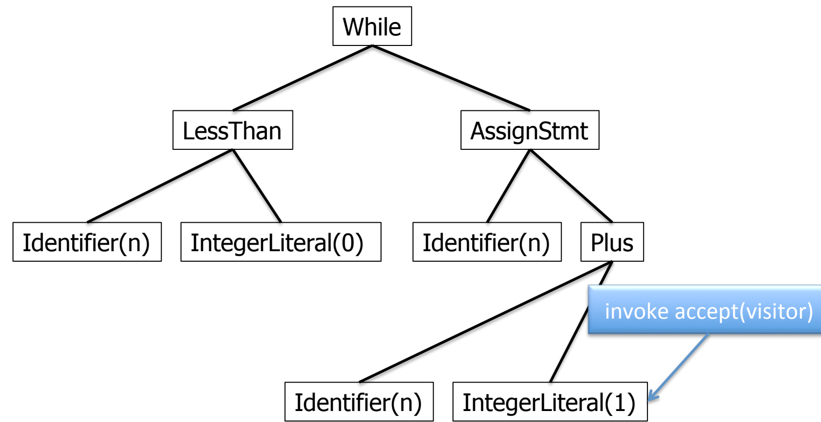
Visitor Pattern, Illustrated

(Assume depth-first traversal)



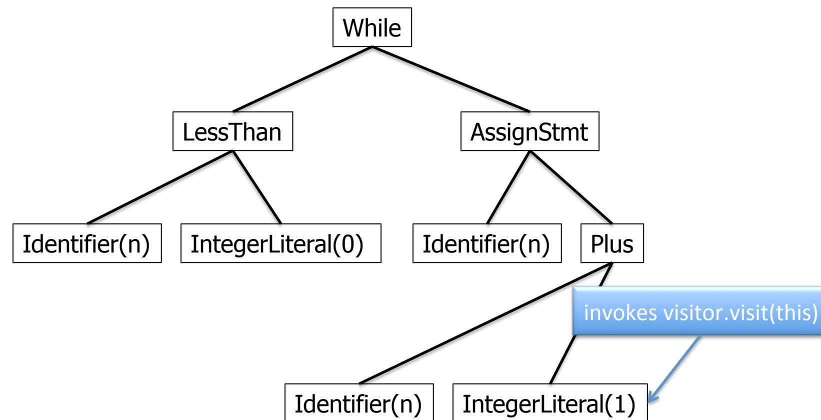
Visitor Pattern, Illustrated

(Assume depth-first traversal)



Visitor Pattern, Illustrated

(Assume depth-first traversal)



Avoiding instanceof

- We'd like to avoid huge if-elseif nests in the visitor to discover the node types

```
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

```
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)
  - Invoked from within a node of the right type, so passing "this" selects correct Visitor method for this node
  - A.k.a., "Double dispatch"

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, although that is common

Accept Method in Each 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);   // 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

Composite Objects

- What if an AST node refers to subnodes?
- Visitors often control the traversal
  - Why? The "~~father~~ visitor knows best" what kind of traversal (if any) it needs)

```
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 Visitor

- Let's check out src/AST/Visitor/
- There's a pretty print visitor (prints out something that looks like a decompiled version of the code).
  - Note: this is **not** what the output of your project parse tree printer should look like.

Encapsulation

- A visitor object often needs to be able to access state in the AST nodes
  - ∴ 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)

Visitor Actions

- A visitor function has a reference to the node it is visiting (the parameter)
  ∴ can access and manipulate subtrees directly

- Visitor object can also include local data (state) shared by the visitor methods

```
public class TypeCheckVisitor extends NodeVisitor {
    public void visit(WhileNode s) { … }
    public void visit(IfNode s) { … }
    …
    private <local state>;  // all methods can read/write this
}
```

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)
  - *Object-Oriented Design & Patterns*, Horstmann, A-W, 2nd ed, 2006 (uses Java)
- Specific information for MiniJava AST and visitors in Appel textbook & online

Coming Attractions

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