# CS412/CS413

# Introduction to Compilers

Tim Teitelbaum

## Lecture 11: Syntax-Directed Definitions

February 13, 2008

# Parsing Techniques

- LL parsing
  - Computes a Leftmost derivation
  - Determines the derivation top-down
  - LL parsing table indicates which production to use for expanding the leftmost non-terminal
- LR parsing
  - Computes a Rightmost derivation
  - Determines the derivation bottom-up
  - Uses a set of LR states and a stack of symbols
  - LR parsing table indicates, for each state, what action to perform (shift/reduce) and what state to go to next
- Use these techniques to construct an AST

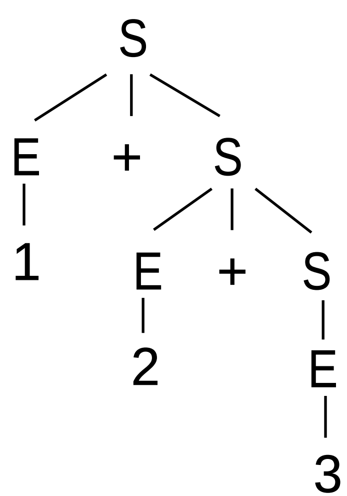
# AST Review

- Derivation = sequence of applied productions

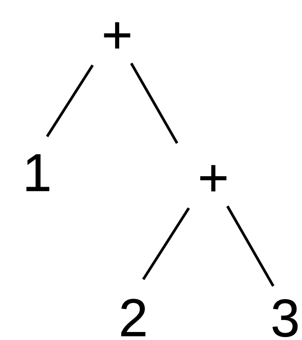
  $S \Rightarrow E + S \Rightarrow 1 + S \Rightarrow 1 + E \Rightarrow 1 + 2$

- Parse tree = graph representation of a derivation
  - Doesn't capture the order of applying the productions
- Abstract Syntax Tree (AST) discards unnecessary information from the parse tree

Parse Tree

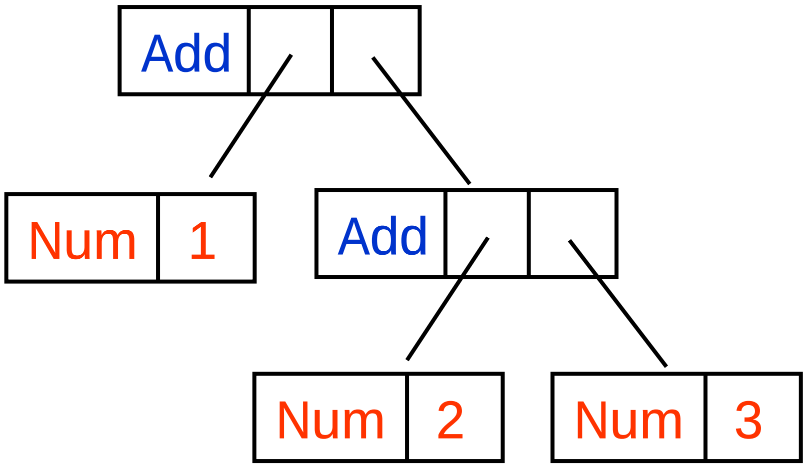



AST



# AST Data Structures

```
abstract class Expr { }

class Add extends Expr {
    Expr left, right;
    Add(Expr L, Expr R) {
        left = L; right = R;
    }
}

class Num extends Expr {
    int value;
    Num (int v) { value = v; }
}
```



# AST Construction

- LL/LR parsing implicitly walks parse tree during parsing
  - LL parsing: Parse tree implicitly represented by sequence of derivation steps (preorder)
  - LR parsing: Parse tree implicitly represented by sequence of reductions (endorder)
- The AST is implicitly defined by parse tree
- Want to explicitly construct AST during parsing:
  - add code to parser to build it

# LL AST Construction

- LL parsing: extend procedures for nonterminals
- Example:

$$
\begin{aligned}
S &\rightarrow ES' \\
S' &\rightarrow \varepsilon \mid + S \\
E &\rightarrow \text{num} \mid ( S )
\end{aligned}
$$

```
void parse_S() {
  switch (token) {
    case num: case '(':
      parse_E();
      parse_S'();
      return;
    default:
      throw new ParseError();
  }
}
```

```
Expr parse_S() {
  switch (token) {
    case num: case '(':
      Expr left = parse_E();
      Expr right = parse_S'();
      if (right == null) return left;
      else return new Add(left, right);
    default: throw new ParseError();
  }
}
```

# LR AST Construction

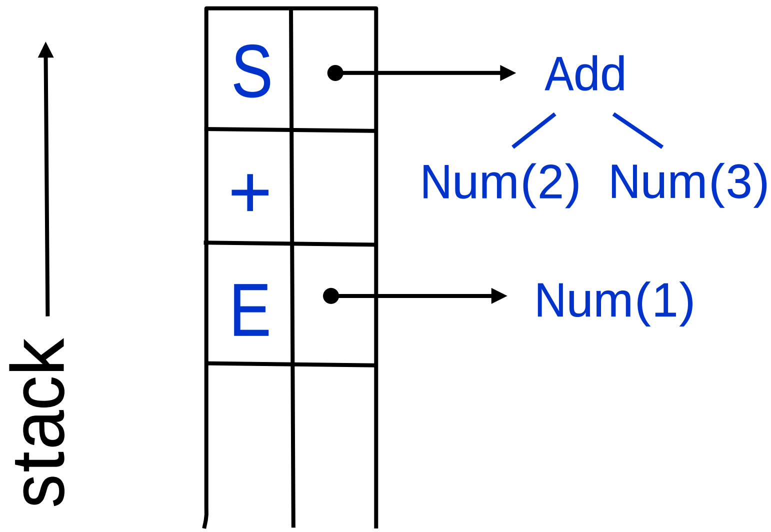
- LR parsing
  - Need to add code for explicit AST construction

- AST construction mechanism for LR Parsing
  - With each symbol X on stack, also store AST sub-tree for X on stack
  - When parser performs reduce operation for $A \to \beta$, create AST subtree for A from AST fragments on stack for $\beta$, pop $|\beta|$ subtrees from stack, push subtree for $\beta$.

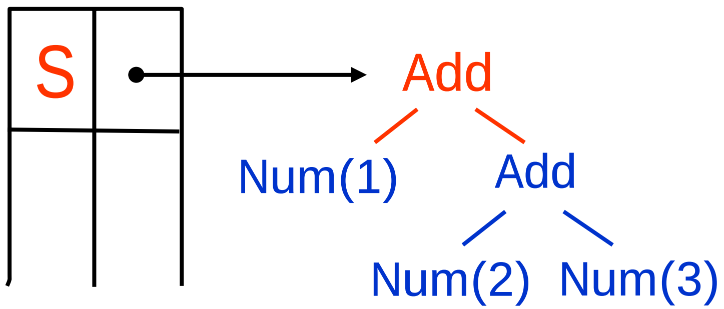
# LR AST Construction, ctd.

- Example

$S \to E+S \mid S$
$E \to num \mid ( S )$




Before reduction
$S \to E+S$




After reduction
$S \to E+S$

# Issues

- Unstructured code: mixed parsing code with AST construction code
- Automatic parser generators
  - The generated parser needs to contain AST construction code
  - How to construct a customized AST data structure using an automatic parser generator?
- May want to perform other actions concurrently with the parsing phase
  - E.g., semantic checks
  - This can reduce the number of compiler passes

# Syntax-Directed Definition

- Solution: syntax-directed definition
  - Extends each grammar production with an associated semantic action (code):

$$S \rightarrow E{+}S \qquad \{\ \text{action}\ \}$$

  - The parser generator adds these actions into the generated parser
  - Each action is executed when the corresponding production is reduced

# Semantic Actions

- Actions = code in a programming language
  - Same language as the automatically generated parser
- Examples:
  - Yacc = actions written in C
  - CUP = actions written in Java
- The actions can access the parser stack!
  - Parser generators extend the stack of states (corresponding to RHS symbols) symbols with entries for user-defined structures (e.g., parse trees)
- The action code need to refer to the states (corresponding to the RHS grammar symbols) in the production
  - Need a naming scheme…

# Naming Scheme

- Need names for grammar symbols to use in the semantic action code
- Need to refer to multiple occurrences of the same nonterminal symbol

$$E \rightarrow E_1 + E_2$$

- Distinguish the nonterminal on the LHS

$$E_0 \rightarrow E + E$$

# Naming Scheme: CUP

- CUP:
  - Name RHS nonterminal occurrences using distinct, user-defined labels:

    expr ::= expr:e1 PLUS expr:e2

  - Use keyword RESULT for LHS nonterminal

- CUP Example (an interpreter):

```
expr ::= expr:e1 PLUS expr:e2
        {:   RESULT = e1 + e2;   :}
```

# Naming Scheme: yacc

- Yacc:
  - Uses keywords: $1 refers to the first RHS symbol, $2 refers to the second RHS symbol, etc.
  - Keyword $$ refers to the LHS nonterminal

- Yacc Example (an interpreter):

```
expr ::= expr PLUS expr    { $$ = $1 + $3; }
```

# Building the AST

- Use semantic actions to build the AST
- AST is built bottom-up during parsing

User-defined type for semantic objects on the stack

Nonterminal name

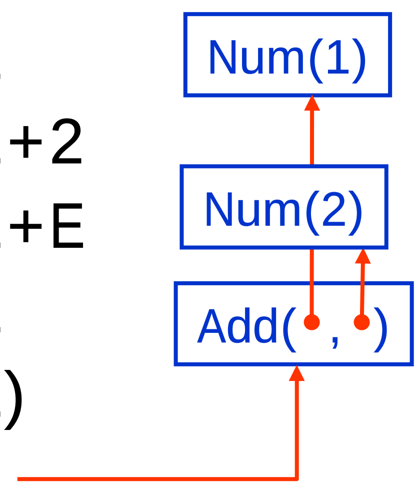
```
non terminal Expr expr;

expr ::= NUM:i                    {: RESULT = new Num(i.val); :}
expr ::= expr:e1 PLUS expr:e2     {: RESULT = new Add(e1,e2); :}
expr ::= expr:e1 MULT expr:e2     {: RESULT = new Mul(e1,e2); :}
expr ::= LPAR expr:e RPAR         {: RESULT = e; :}
```

# Example

$\boxed{E \rightarrow \text{num} \mid (E) \mid E{+}E \mid E{*}E}$

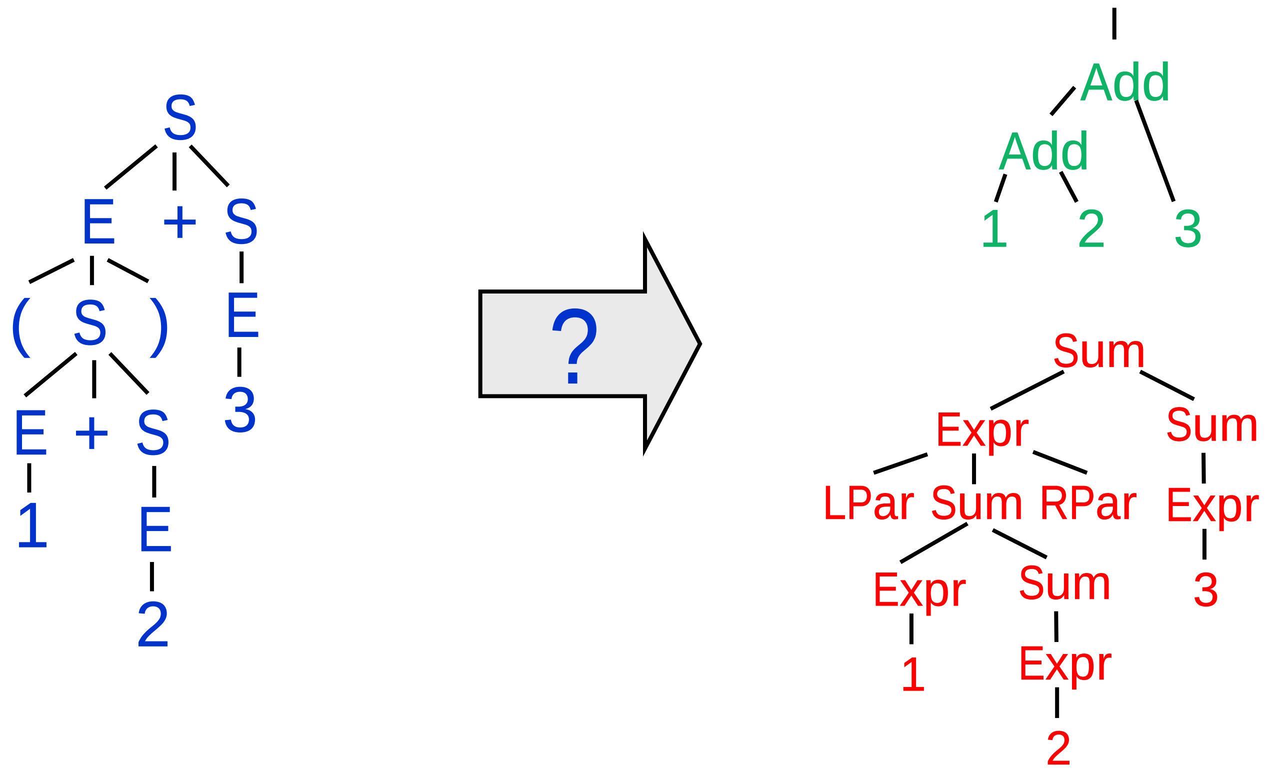
- Parser stack stores value of each symbol

| | | |
|---|---|---|
| | (1+2)*3 | |
| (1 | +2)*3 | |
| (E | +2)*3 | RESULT=new Num(1) |
| (E+2 | )*3 | |
| (E+E | )*3 | RESULT=new Num(2) |
| (E | )*3 | RESULT=new Add(e1,e2) |
| (E) | *3 | |
| E | *3 | RESULT=e |



# AST Design

- Keep the AST abstract
- Do not introduce tree node for every node in parse tree (not very abstract)



# AST Design

- Do not use single class AST_node
- E.g., need information for if, while, +, *, ID, NUM

```
class AST_node {
    int node_type;
    AST_node[ ] children;
    String name; int value; ...etc...
}
```

- Problem: must have fields for every different kind of node with attributes
- Not extensible, Java type checking no help

# Use Class Hierarchy

- Use subclassing to solve problem
  - Use abstract class for each “interesting” set of nonterminals (e.g., expressions)

$$E \rightarrow E+E \mid E*E \mid -E \mid (E)$$

```
abstract class Expr { ... }
class Add extends Expr { Expr left, right; ... }
class Mult extends Expr { Expr left, right; ... }
// or: class BinExpr extends Expr { Oper o; Expr l, r;  }
class Minus extends Expr { Expr e; ...}
```

# Another Example

E ::= num | (E) | E+E | id

S ::= E ; | if (E) S |
    if (E) S else S | id = E ; | ;

```
abstract class Expr { ... }
class Num extends Expr { Num(int value) ... }
class Add extends Expr { Add(Expr e1, Expr e2) ... }
class Id extends Expr { Id(String name) ... }

abstract class Stmt { ... }
class IfS extends Stmt { IfS(Expr c, Stmt s1, Stmt s2) }
class EmptyS extends Stmt { EmptyS() ... }
class AssignS extends Stmt { AssignS(String id, Expr e)...}
```

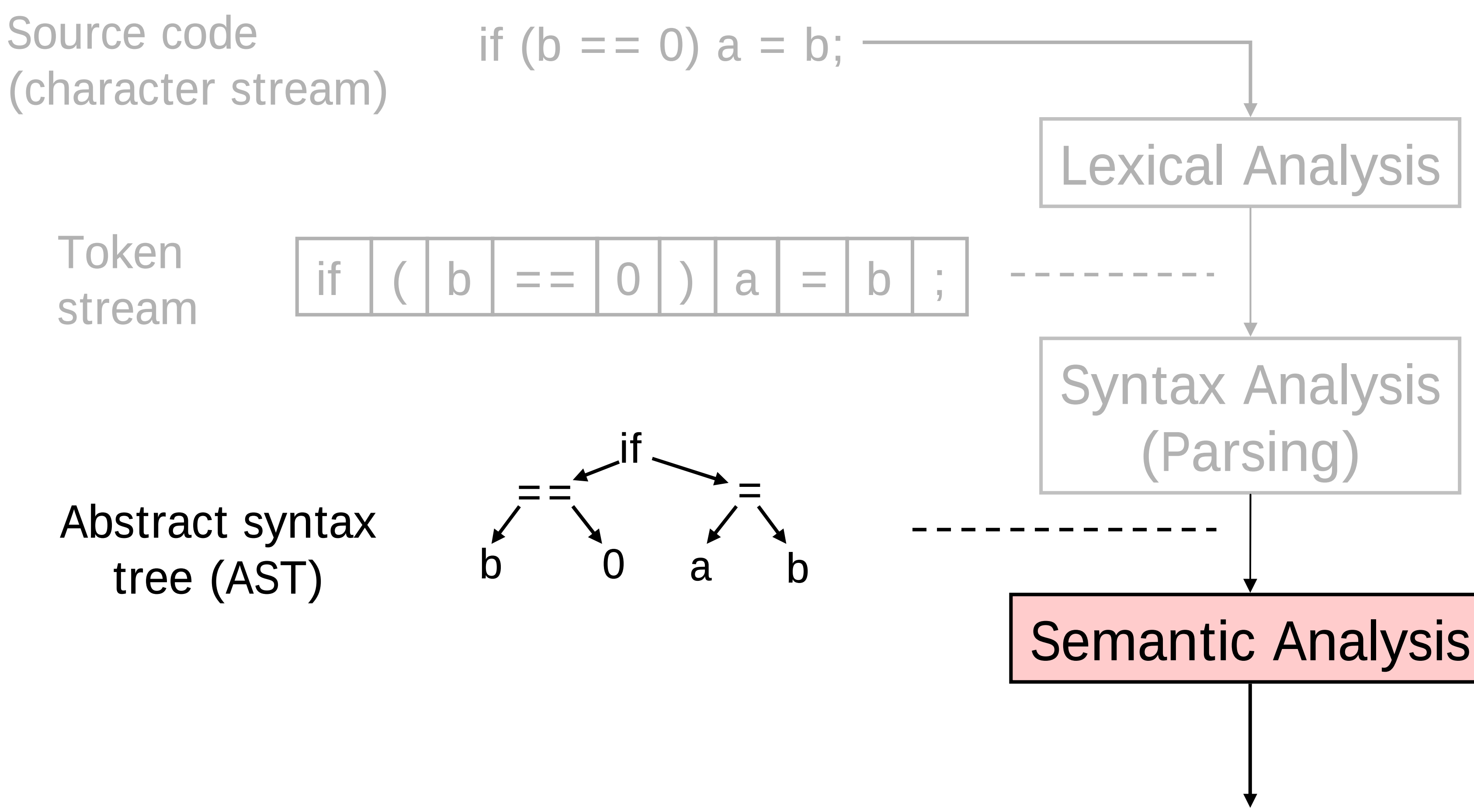
# Other Syntax-Directed Definitions

- Can use syntax-directed definitions to perform semantic checks during parsing
  - E.g., type-checking
- Benefit = efficiency
  - One compiler pass for multiple tasks
- Disadvantage = unstructured code
  - Mixes parsing and semantic checking phases
  - Perform checks while AST is changing
  - Limited to one pass in bottom-up order

# Structured Approach

- Separate AST construction from semantic checking phase
- Traverse AST and perform semantic checks (or other actions) only after tree has been built and its structure is stable
- Approach is more flexible and less error-prone
  - It is better when efficiency is not a critical issue

# Where We Are


Source code
(character stream)
if (b == 0) a = b;
Lexical Analysis
Token
stream
if
(
b
==
0
)
a
=
b
;
Syntax Analysis
(Parsing)
Abstract syntax
tree (AST)
if
==
=
b
0
a
b
Semantic Analysis

# AST Data Structure

```
abstract class Expr {



    }
class Add extends Expr { ...
    Expr e1, e2;



    }
class Num extends Expr { ...
    int value;



    }
class Id extends Expr { ...
    String name;


    }
```

# Could add AST computation to class, but...

```
abstract class Expr {
    ...; /* state variables for visitA */

    }
class Add extends Expr { ...
    Expr e1, e2;
    void visitA(){ ...; visitA(this.e1); ...; visitA(this.e2); ...}

    }
class Num extends Expr { ...
    int value;
    void visitA(){...}

    }
class Id extends Expr { ...
    String name;
    void visitA(){...}

    }
```

# Undesirable Approach to AST Computation

```
abstract class Expr {
    ...; /* state variables for visitA */
    ...; /* state variables for visitB */
    }
class Add extends Expr { ...
    Expr e1, e2;
    void visitA(){ ...; visitA(this.e1); ...; visitA(this.e2); ...}
    void visitB(){ ...; visitB(this.e2); ...; visitB(this.e1); ...}
    }
class Num extends Expr { ...
    int value;
    void visitA(){...}
    void visitB(){...}
    }
class Id extends Expr { ...
    String name;
    void visitA(){...}
    void visitB(){...}
    }
```

# Visitor Methodology for AST Traversal

- Visitor pattern: separate data structure definition (e.g., AST) from algorithms that traverse the structure (e.g., name resolution code, type checking code, etc.).
- Define Visitor interface for all AST traversals
- Extend each AST class with a method that accepts any Visitor (by calling it back)
- Code each traversal as a separate class that implements the Visitor interface

# Visitor Interface

```
interface Visitor {
  void visit(Add e);
  void visit(Num e);
  void visit(Id e);
}
```

# Accept methods

```
abstract class Expr { ...
    abstract public void accept(Visitor v);
    }
class Add extends Expr { ...
    public void accept(Visitor v) {
        v.visit(this); }
    }
class Num extends Expr { ...
    public void accept(Visitor v) {
        v.visit(this); }
    }
class Id extends Expr { ...
    public void accept(Visitor v) {
        v.visit(this); }
    }
```

The declared type of **this** is the subclass in which it occurs.

Overload resolution of

v.visit(**this**);

invokes appropriate visit function in Visitor v.

# Visitor Methods

- For each kind of traversal, implement the Visitor interface, e.g.,

```
class PostfixOutputVisitor implements Visitor {
    void visit(Add e)  {
        e.e1.accept(this); e.e2.accept(this); System.out.print( "+" );
        }
    void visit(Num e)   {
        System.out.print(e.value);
        }
    void visit(Id e)   {
        System.out.print(e.id);
        }
}
```

> Dynamic dispatch e'.accept invokes accept method of appropriate AST subclass and eliminates case analysis on AST subclasses

- To traverse expression e:

```
    PostfixOutputVisitor v = new PostfixOutputVisitor();
    e.accept(v);
```

# Inherited and Synthesized Information

- So far, OK for traversal and action w/o communication of values
- But we need a way to pass information
  - Down the AST (inherited)
  - Up the AST (synthesized)
- To pass information down the AST
  - add parameter to visit functions
- To pass information up the AST
  - add return value to visit functions

# Visitor Interface (2)

```
interface Visitor {
  Object visit(Add e, Object inh);
  Object visit(Num e, Object inh);
  Object visit(Id e, Object inh);
}
```

# Accept methods (2)

```
abstract class Expr { ...
    abstract public Object accept(Visitor v, Object inh);
    }
class Add extends Expr { ...
    public Object accept(Visitor v, Object inh) {
        return v.visit(this, inh); }
     }
class Num extends Expr { ...
    public Object accept(Visitor v, Object inh) {
         return v.visit(this, inh);
    }
class Id extends Expr { ...
    public Object accept(Visitor v, Object inh) {
         return v.visit(this, inh); }
    }
}
```

# Visitor Methods (2)

- For each kind of traversal, implement the Visitor interface, e.g.,

```
class EvaluationVisitor implements Visitor {
    Object visit(Add e, Object inh)  {
        int left = (int) e.e1.accept(this, inh);
        int right = (int) e.e2.accept(this, inh);
        return left+right;
        }
    Object visit(Num e, Object inh)   {
        return value;
        }
    Object visit(Id e, Object inh)   {
        return Lookup(id, (SymbolTable)inh);
        }
}
```

- To traverse expression e:

```
    EvaluationVisitor v = new EvaluationVisitor ();
    e.accept(v, EmptyTable());
```

# Summary

- Syntax-directed definitions attach semantic actions to grammar productions
- Easy to construct the AST using syntax-directed definitions
- Can use syntax-directed definitions to perform semantic checks, but better not to
- Separate AST construction from semantic checks or other actions that traverse the AST