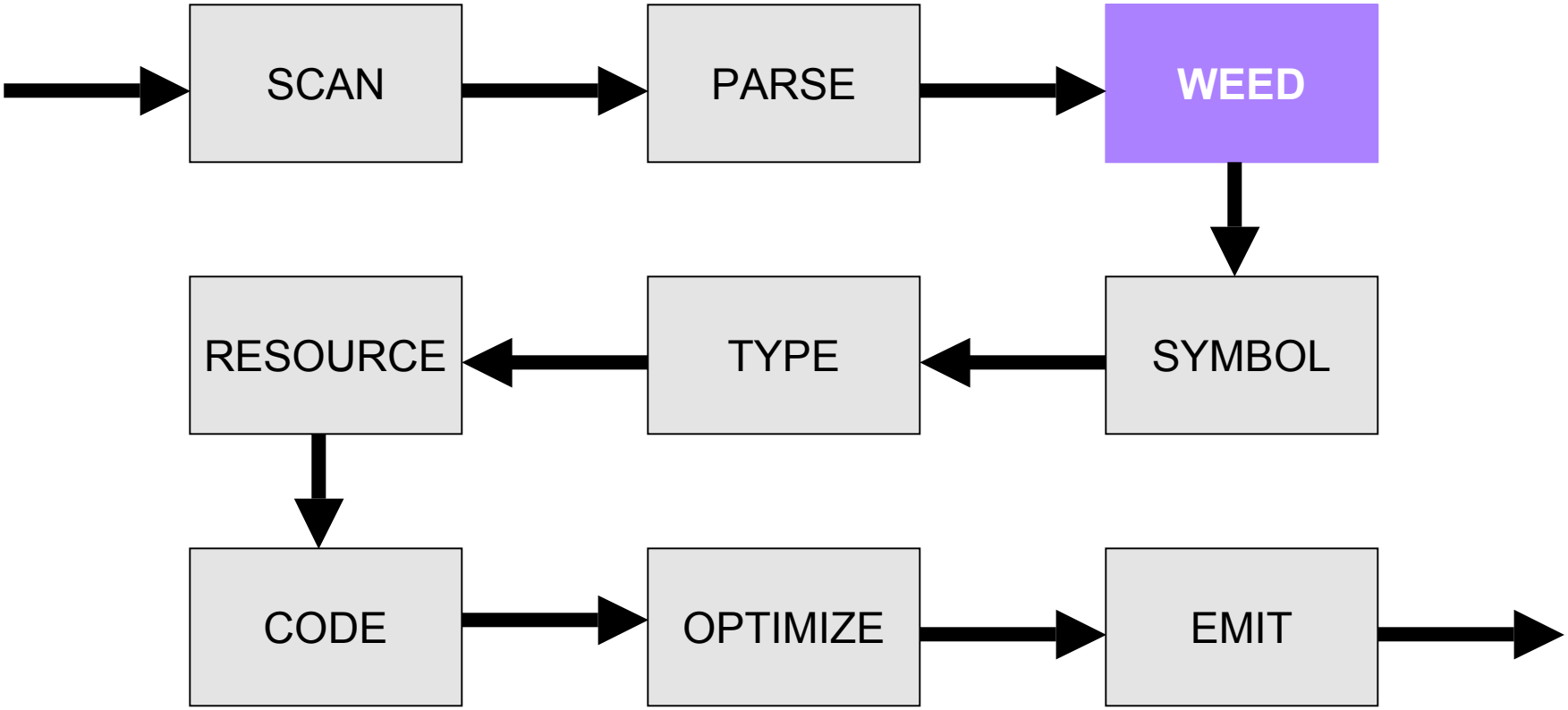
Compiler

Abstract Syntax Trees

Compiler Architecture



Phases and Passes

- A compiler *phase* is a cohesive functional unit that processes a representation of the source program
- A compiler *pass* is a traversal of the source program representation
- Multiple phases can be integrated into a single pass
  - e.g., parser-driven scanning

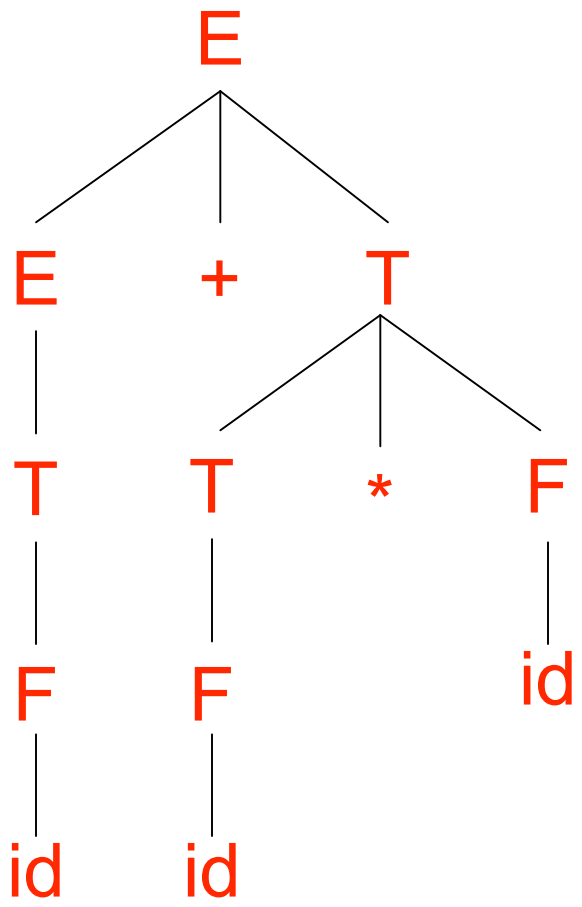
Single Pass Compilation

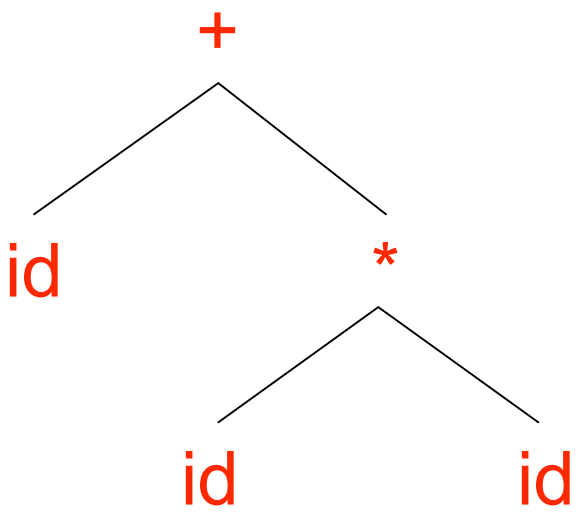
- All processing must happen in a pipelined fashion
  - Restricts languages (forward declarations)
  - Limits optimization
- Used to be popular:
  - fast (if your machine is slow); and
  - space efficient (if you only have 4K RAM)
- A modern compiler uses 5-15 passes

Abstract Syntax Trees

- Common representation for program
  - Represents all of the semantic entities in the program
  - Eliminates irrelevant syntactic details
- Can be thought of as a compressed parse tree
  - Need extra structure in the grammar during parsing for precedence, etc.
  - Don’t need all of that structure subsequently

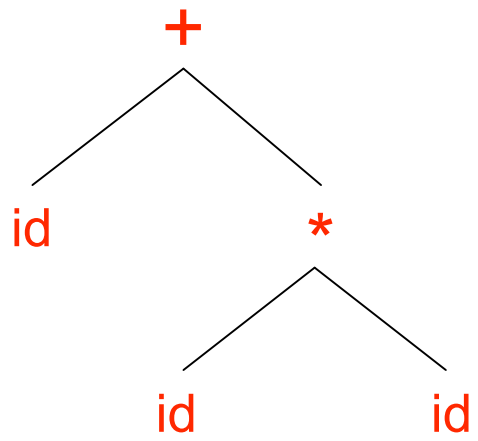
Parse Tree vs. AST





Intermediate Languages

- ASTs can be thought of as the input or output language of a phase
  - e.g., parser outputs ASTs, code generator takes AST as input
- *Linear forms* for ASTs allow phases to be written as separate programs
  - With their own parsers, etc.



+(id,*(id,id))

Expression AST

- Enhance expression parser to produce ASTs
- Need to design
  - An in-memory AST data structure
  - A linearized external form (for debugging purposes mainly)
- We'll study a small ANTLR example
  - Source is online for you to play with

AST Data Structure

- ANTLR provides a default AST structure
  - Allows arbitrary children using left-most child/right-sibling organization
  - Enabled by parser option `buildAST`
  - Will construct something like the parse tree by default
- We'll build an expression AST
  - Abstract base type
  - Binary operator sub-type
  - Operator specific sub-types
  - Literal specific leaves

ExprAST

```
package antlr.example.explicitast;

public abstract class ExprAST extends antlr.BaseAST
{
  public abstract int value();
  public abstract String pretty();
}
```

BinaryOperatorAST

```
package antlr.example.explicitast;

public abstract class BinaryOperatorAST extends ExprAST
{
    public ExprAST left()
    {
        return (ExprAST) getFirstChild();
    }

    public ExprAST right()
    {
        ExprAST t = left();
        assert t != null;
        return (ExprAST) t.getNextSibling();
    }
}
```

PlusNodeAST

```
…
public class PlusNodeAST extends BinaryOperatorAST {
    public PlusNodeAST(Token tok) {
    }

    public int value() {
        return left().value() + right().value();
    }

    public String pretty() {
        return "(" + left().pretty() + " + " +
               right().pretty() + ")";
    }
…
}
```

LiteralAST

```
public class LiteralAST extends ExprAST {
  int v = 0;

  public LiteralAST(Token tok) {
    v = Integer.parseInt(tok.getText());
  }

  public int value() {
    return v;
  }

  public String pretty() {
    return ""+v;
  }
...
}
```

Syntax-directed Translation

- We will use the structure of the grammar to define how trees are built up
  - The order of rule matching/reduction will determine the sequence of AST constructor calls
- Need to declare the AST types in the parser specification in order to
  - Pass AST nodes up the parser call stack
  - Call AST methods

Builtin vs. External ASTs

- We are using the ANTLR builtin AST support
  - SJC uses an external AST
- We will use ANTLR notation to indicate
  - AST types associated with parse actions
  - Which parse nodes should be included in the AST
- For SJC we state all of that explicitly

AST Declaration

```
tokens
{
  PLUS<AST=antlr.example.explicitast.PlusNodeAST>;

  MINUS<AST=antlr.example.explicitast.MinusNodeAST>;

  TIMES<AST=antlr.example.explicitast.TimesNodeAST>;

  INT<AST=antlr.example.explicitast.LiteralAST>;
}
```

Parse Actions

```
expr:   mexpr ((PLUS^|MINUS^) mexpr)*
    ;

mexpr
    :   atom (TIMES^ atom)*
    ;

atom:   INT^
    |   LPAREN! expr RPAREN!
    ;
```

ANTLR Rule Directives

- The ^ in a rule indicates the token used to determine the AST node that is built
  - The logic of the constructor/initializer actually assembles the AST from ASTs of other production elements
- The ! In a rule indicates that the token should be ignored for the purpose of ASTs
  - We effectively discard () here after using them to enforce parse structure

Enriching ASTs

- ASTs can carry lots of information about a program and its sub-parts
  - phases/passes may add their own info
- Scanner → line numbers
  - Useful for error messages
- Symbol processing → identifier meaning
- Type checker → expression types
- Code Generation → assembler code

Enriched AST Data Structure

- Trivial to add this to our AST by extending the base type ExprAST with new fields
  - Updating the constructor/intializer to, e.g., add line number information collected by the lexer

For You To Do

- Extend the lexer to keep track of source line number
- Extend AST to record line number information
- Extend pretty printer to dump line number information
  - NB: this will break the existing unit tests

External Form

- Could produce pre-fix representation
- We will produce an infix representation that fully-parenthesizes source expression
  - i.e., a pretty printer
- We exploit AST visitor structure
  - Implement pretty() function in each node

Putting it all together

```
public class ExprTreeTest extends TestCase {
  public void testNoParen() {
    testPass("1 + 3 * 2", 7, "(1 + (3 * 2))");
  }
  …
  public static void testPass(String i, int r, String p)
  {
    try {
      …
    } catch (ANTLRException e) {
      e.printStackTrace();
      Assert.assertTrue(e.getMessage(), false);
    }
  }
}
```

Putting it all together

```
ExprLexer lexer = new ExprLexer(new StringReader(in));
ExprTreeParser parser = new ExprTreeParser(lexer);
ExprAST ast = (ExprAST)(parser.expr().getAST());

Assert.assertTrue(ast.value() == r);

String s = ast.pretty();
Assert.assertTrue(s.equals(p));

System.out.println(s);
```

Front-end Testing

- Each part of the front-end is a little translator
  - Character stream _ token stream
  - Token stream _ AST
  - AST _ AST’
- As long as you have an external form for your data structures you can test them independently or in combination

A Testing Strategy

- Let *parse* be the front-end and *pretty* be the pretty printing pass discussed earlier

Does pretty(parse(P)) = P ?

Eliminating White-space Sensitivity

- An application of *pretty*(*parse*()) eliminates white-space
  - we can just apply it to both sides

pretty(*parse*(*pretty*(*parse*(P)))) = *pretty*(*parse*(P))

Eclipse Java Development Tooling (JDT) Domain Object Model (DOM)

- set of classes (in the `org.eclipse.jdt.core.dom` package) that model the source code of a Java program as a structured document
- has nodes for each syntactic category in Java
- each represented by its own structure
  - created by using the **`AST`** factory class
  - top-level **`ASTNode`** class provides utility methods, for exampe, to copy (deep-clone) sub-trees
- extensively documented!

StaticJava in JDT DOM

StaticJava	Eclipse JDT DOM Class
<*program*>	CompilationUnit
<*class-declaration*>	TypeDeclaration
<*main-method-declaration*>	MethodDeclaration
<*field-declaration*>	FieldDeclaration
<*method-declaration*>	MethodDeclaration
<*type*>	Type (PrimitiveType)
<*param*>	SingleVariableDeclaration
<*method-body*>	Block
<*local-declaration*>	VariableDeclarationStatement
…	…

CompilationUnit

Overview Package Class Use Tree Deprecated Index Help

PREV CLASS NEXT CLASS FRAMES NO FRAMES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

org.eclipse.jdt.core.dom

Class CompilationUnit

```
java.lang.Object
  └ org.eclipse.jdt.core.dom.ASTNode
      └ org.eclipse.jdt.core.dom.CompilationUnit
```

public class **CompilationUnit**
extends ASTNode

Java compilation unit AST node type. This is the type of the root of an AST.

The source range for this type of node is ordinarily the entire source file, including leading and trailing whitespace and comments.

For JLS2:

```
CompilationUnit:
   [ PackageDeclaration ]
       { ImportDeclaration }
       { TypeDeclaration | ; }
```

For JLS3, the kinds of type declarations grew to include enum and annotation type declarations:

```
CompilationUnit:
   [ PackageDeclaration ]
       { ImportDeclaration }
       { TypeDeclaration | EnumDeclaration | AnnotationTypeDeclaration | ; }
```

CompilationUnit API

Method Summary	
ASTNode	findDeclaringNode(IBinding binding) Finds the corresponding AST node in the given compilation unit from which the given binding originated.
ASTNode	findDeclaringNode(String key) Finds the corresponding AST node in the given compilation unit from which the binding with the given key originated.
List	getCommentList() Returns a list of the comments encountered while parsing this compilation unit.
int	getExtendedLength(ASTNode node) Returns the extended source length of the given node.
int	getExtendedStartPosition(ASTNode node) Returns the extended start position of the given node.
IJavaElement	getJavaElement() The Java element (an org.eclipse.jdt.core.ICompilationUnit or an org.eclipse.jdt.core.IClassFile) this compilation unit was created from, or null if it was not created from a Java element.
PackageDeclaration	getPackage() Returns the node for the package declaration of this compilation unit, or null if this compilation unit is in the default package.
List	imports() Returns the live list of nodes for the import declarations of this compilation unit, in order of appearance.
int	lineNumber(int position) Returns the line number corresponding to the given source character position in the original source string.
static List	propertyDescriptors(int apiLevel) Returns a list of structural property descriptors for this node type.
void	recordModifications() Enables the recording of changes to this compilation unit and its descendents.
TextEdit	rewrite(IDocument document, Map options) Converts all modifications recorded for this compilation unit into an object representing the corresponding text edits to the given document containing the original source code for this compilation unit.
void	setPackage(PackageDeclaration pkgDecl) Sets or clears the package declaration of this compilation unit node to the given package declaration node.
List	types() Returns the live list of nodes for the top-level type declarations of this compilation unit, in order of appearance.

CompilationUnit Example

```
// create a Java AST factory object (Java Lang Spec 3)
AST ast = AST.newAST(AST.JLS3);

// create a compilation unit
CompilationUnit cu = ast.newCompilationUnit();

// add a new class
{
    TypeDeclaration td = ast.newTypeDeclaration();

    // unfortunately, JDT DOM doesn't use generic
    // i.e., no type check when adding new element
    // but runtime check is done by the API
    cu.types().add(td); // what is td's class name?
}

// iterate over the types in the compilation unit
for (Object o: cu.types())
{
    TypeDeclaration td = (TypeDeclaration) o;

    // process td
}
```

JDT ASTViewer Plugin (Demo)

```
// create a Java AST factory object (Java Lang Spec 3)
AST ast = AST.newAST(AST.JLS3);

// create a compilation unit
CompilationUnit cu = ast.newCompilationUnit();

// add a new class
{
    TypeDeclaration td = ast.newTypeDeclaration();

    // unfortunately, JDT DOM doesn't use generic
    // i.e., no type check when adding new element
    // but runtime check is done by the API
    cu.types().add(td); // what is td's class name?
}

// iterate over the types in the compilation unit
for (Object o: cu.types())
{
    TypeDeclaration td = (TypeDeclaration) o;

    // process td
}
```

JDT DOM Designs

- independent of any JDK version
  - i.e., Java generics cannot be leveraged
- default values for missing node info
  - reduces runtime exception checks
- uses the visitor pattern for AST traversal
  - separates AST traversal from node processing
  - leverages dynamic method dispatch in OOP

Bottom-Up AST Construction in Top-Down Parsing

- Top-down parsers perform actions when they recognize terminals or non-terminals
  - this makes it natural to construct AST structures bottom-up
    - to create AST for **`A`** from the rule **`A : := B`**, then we need to construct **`B`** first
  - in ANTLR we can attach AST construction code in a similar way to the syntax directed evaluation parser example

StaticJava ASTParser _ Compilation Unit

```
compilationUnit
   returns [CompilationUnit result = ast.newCompilationUnit()]
{
  TypeDeclaration td;
}
  : td=classDefinition          { // add a new class
                                  result.types().add(td);
                                }

    EOF
  ;
```

StaticJava ASTParser _ Class Declaration

```
classDefinition returns [TypeDeclaration result = ast.newTypeDeclaration()]
{
  String className; MethodDeclaration md; FieldDeclaration fd;
}
  : "public"                          { result.modifiers().add(
                                            ast.newModifier(Modifier.
                                               ModifierKeyword.PUBLIC_KEYWORD));
                                      }
    "class" id:IDENT LCURLY           { className = id.getText();
                                        result.setName(
                                            ast.newSimpleName(className));
                                      }
    md=mainMethodDeclaration          { result.bodyDeclarations().add(md); }
    (
      ("static" type IDENT SEMI) =>
      fd=fieldDeclaration             { result.bodyDeclarations().add(fd); }
    | md=methodDeclaration            { result.bodyDeclarations().add(md); }
    )* RCURLY
  ;
```

StaticJava ASTParser _ If-Else Statement

```
ifStatement returns [IfStatement result = ast.newIfStatement()]
{
  Block thenStatement = ast.newBlock();
   result.setThenStatement(thenStatement);
  Block elseStatement = ast.newBlock();
   result.setElseStatement(elseStatement);
  Expression condExp; Statement s;
}
  : "if" LPAREN condExp=exp RPAREN{ result.setExpression(condExp); }
    LCURLY
    (
      s=statement                  { thenStatement.statements().add(s); }
    )*
    RCURLY
    ( "else" LCURLY
      (
        s=statement                { elseStatement.statements().add(s); }
      )*
      RCURLY
    )?
  ;
```

StaticJava ASTParser _ Multiplicative Expression

```
multiplicativeExp returns [Expression result = null]
{
  Expression e; InfixExpression.Operator op = null;
}
  : result=unaryExp
    (
      (
        STAR              { op = InfixExpression.Operator.TIMES; }
      | DIV               { op = InfixExpression.Operator.DIVIDE; }
      | MOD               { op = InfixExpression.Operator.REMAINDER; }
      )
      e=unaryExp          {
                            InfixExpression ie = ast.newInfixExpression();
                            ie.setLeftOperand(result);
                            ie.setOperator(op);
                            ie.setRightOperand(e);
                            result = ie;
                          }
    )*
  ;
```

StaticJava ASTParser - Invoke Expression

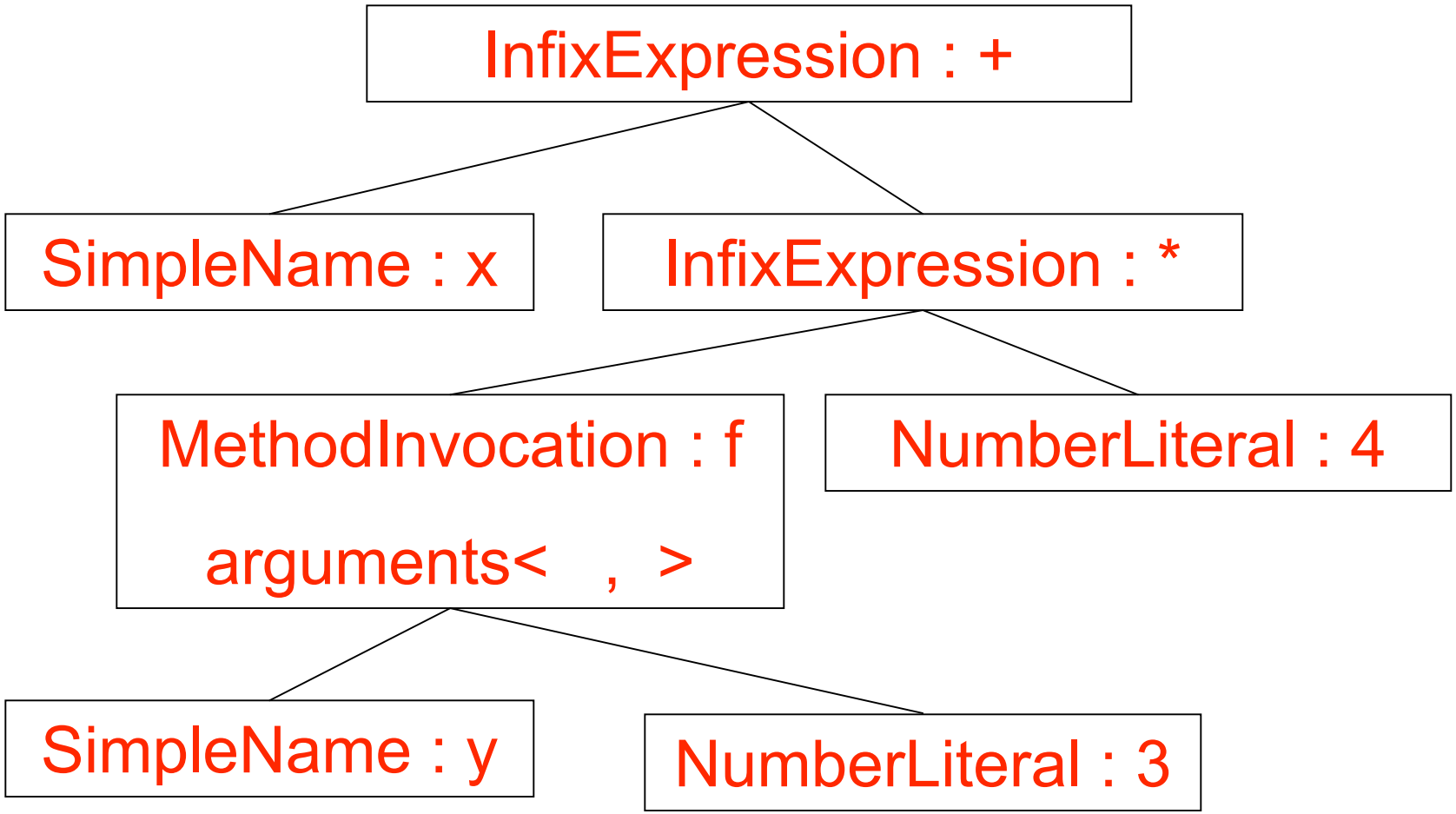
```
invokeExp returns [MethodInvocation result = ast.newMethodInvocation()]
{ String className = null; String methodName; ArrayList<Expression> es; }
  : ( id1:IDENT             { className = id1.getText();
                              result.setExpression(ast.newSimpleName(className));
                            }
      DOT )?
    id2:IDENT               { methodName = id2.getText();
                              result.setName(ast.newSimpleName(methodName));
                            }
    LPAREN ( es=args        { result.arguments().addAll(es); }
    )? RPAREN
  ;

args returns [ArrayList<Expression> result = new ArrayList<Expression>()]
{ Expression e; }
  : e=exp                   { result.add(e); }
    (
      COMMA e=exp           { result.add(e); }
    )*
  ;
```

The Visitor Pattern - Motivation

- Roughly speaking a computation can be broken down into
  - *Data -- the values being operated on*
  - *Algorithm -- the operations applied to the data*
- Many software design approaches advocate coupling data and algorithms
  - *e.g., ADTs usually provide algorithms for encapsulated data behind an opaque interface*
- Sometimes this is the wrong decision
  - *e.g., when an algorithm can be applied to a wide variety of data we don't want to have to implement the common parts of it repeatedly*

x + f(y,3) * 4

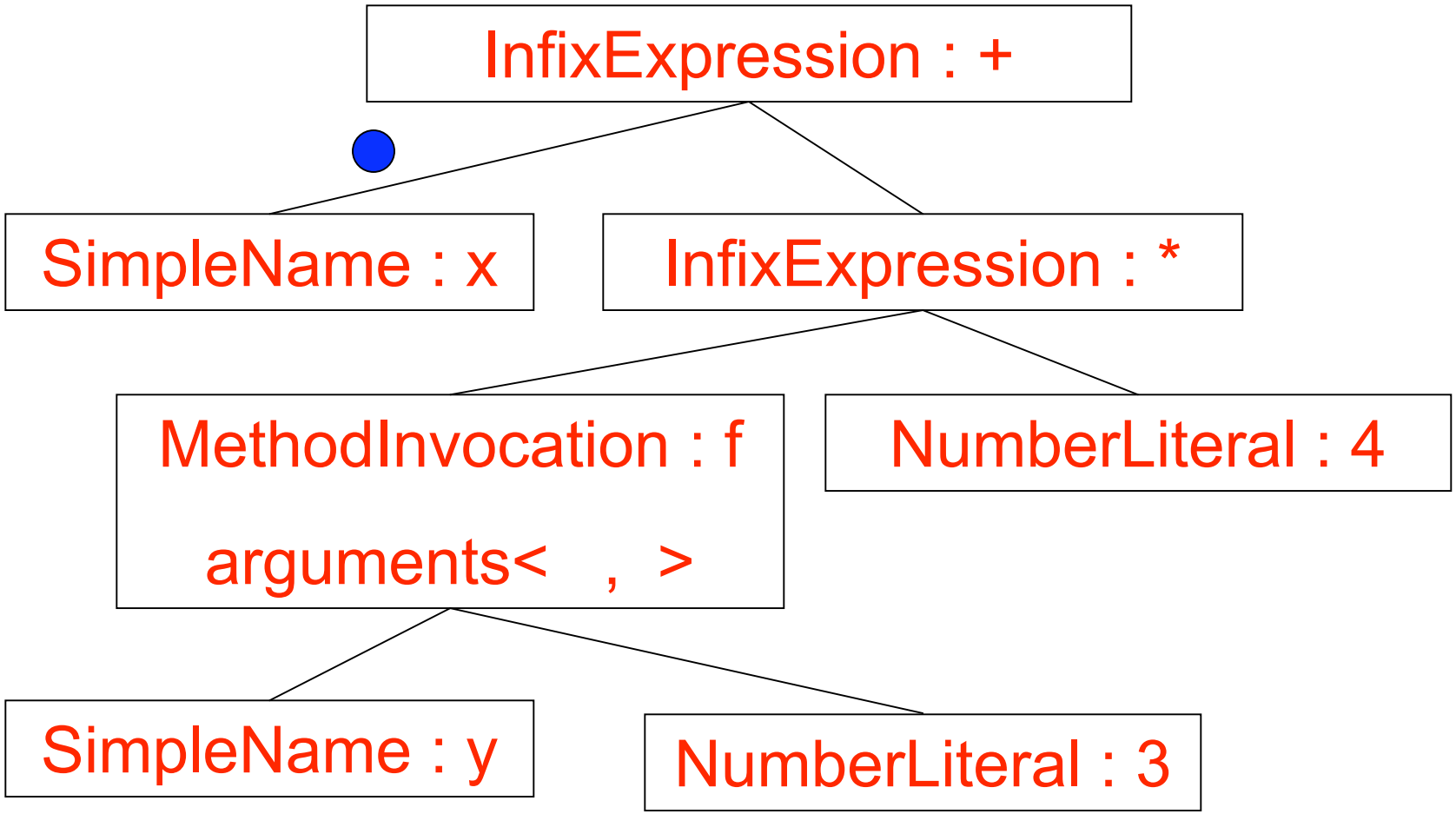

InfixExpression : +
SimpleName : x
InfixExpression : *
MethodInvocation : f
arguments< , >
NumberLiteral : 4
SimpleName : y
NumberLiteral : 3

AST Traversal - Collecting Integer Literals

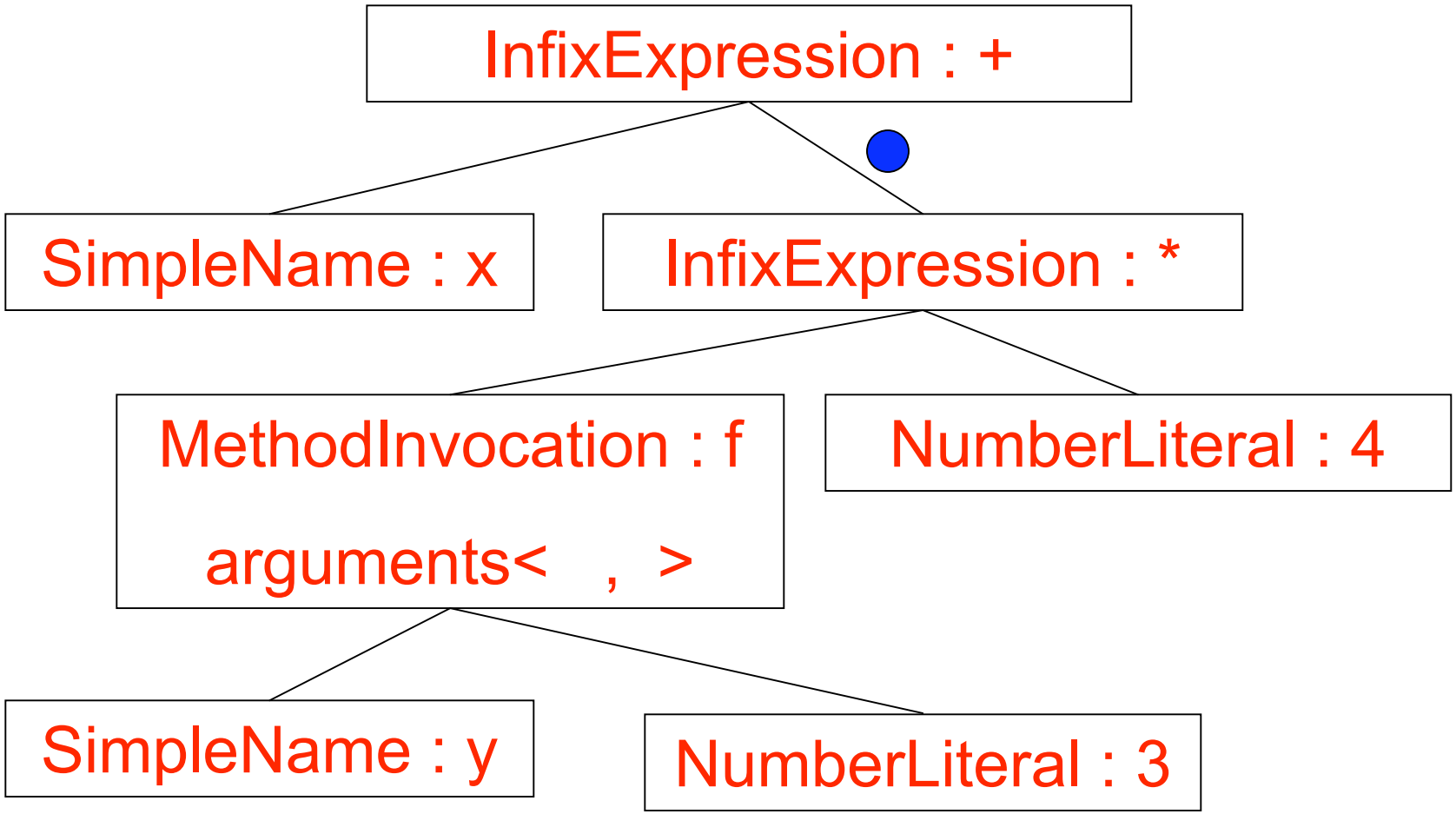
```
public void collectInt(Expression e) {
  if (e instanceof BooleanLiteral) {
    // skip
  } else if (e instanceof NumberLiteral) {
    collectInt((NumberLiteral) e);
  } else if (e instanceof PrefixExpression){
    collectInt((PrefixExpression) e);
  } else if (e instanceof InfixExpression) {
    collectInt((InfixExpression) e);
  } else if (e instanceof
            ParenthesizedExpression) {
    collectInt((ParenthesizedExpression) e);
  } else if (e instanceof
            MethodInvocation) {
    collectInt((MethodInvocation) e);
  } else if (e instanceof SimpleName) {
    // skip
  } else { assert false; } }
```

```
public void collectInt(NumberLiteral e) {
  addInt(Integer.parseInt(e.getToken()));
}
public void collectInt(PrefixExpression e) {
  collectInt(e.getOperand());
}
public void collectInt(InfixExpression e) {
  collectInt(e.getLeftOperand());
  collectInt(e.getRightOperand());
}
public void collectInt(ParenthesizedExpression e)
{
  collectInt(e.getExpression());
}
public void collectInt(MethodInvocation e) {
  for (Object o : e.arguments()) {
    collectInt((Expression) o);
  }
}
```

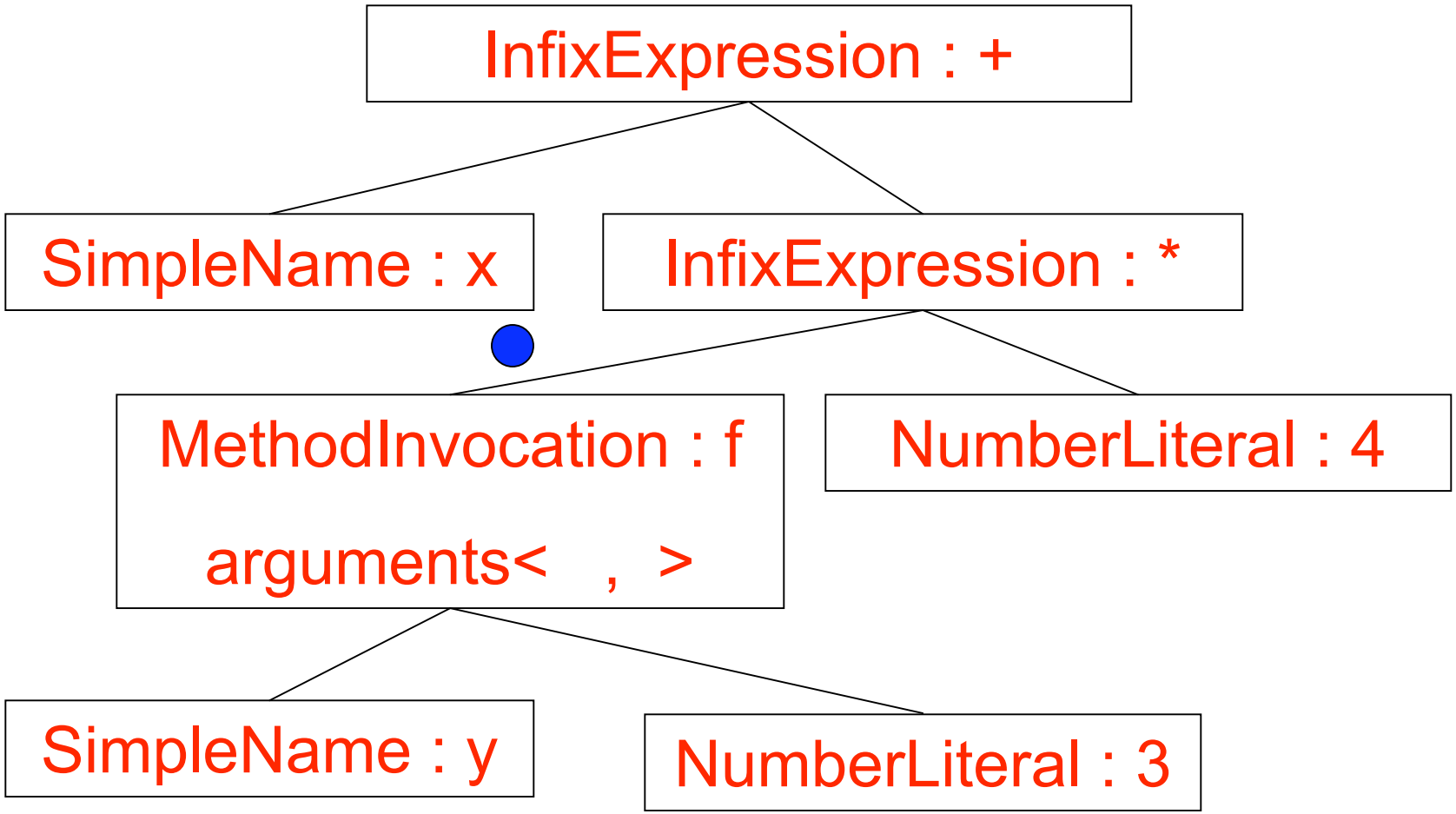
Numbers in x + f(y,3) * 4



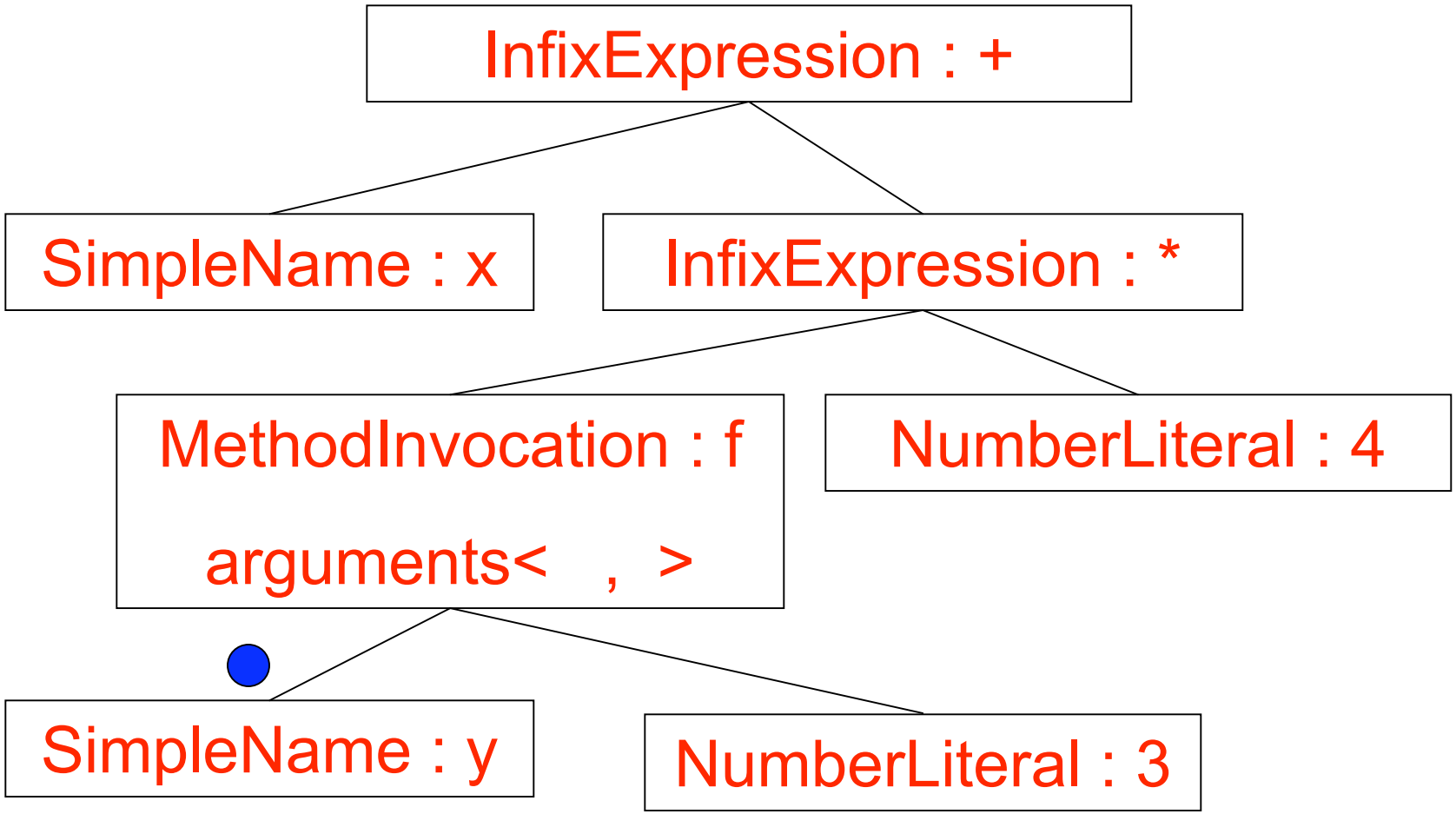
Numbers in x + f(y,3) * 4



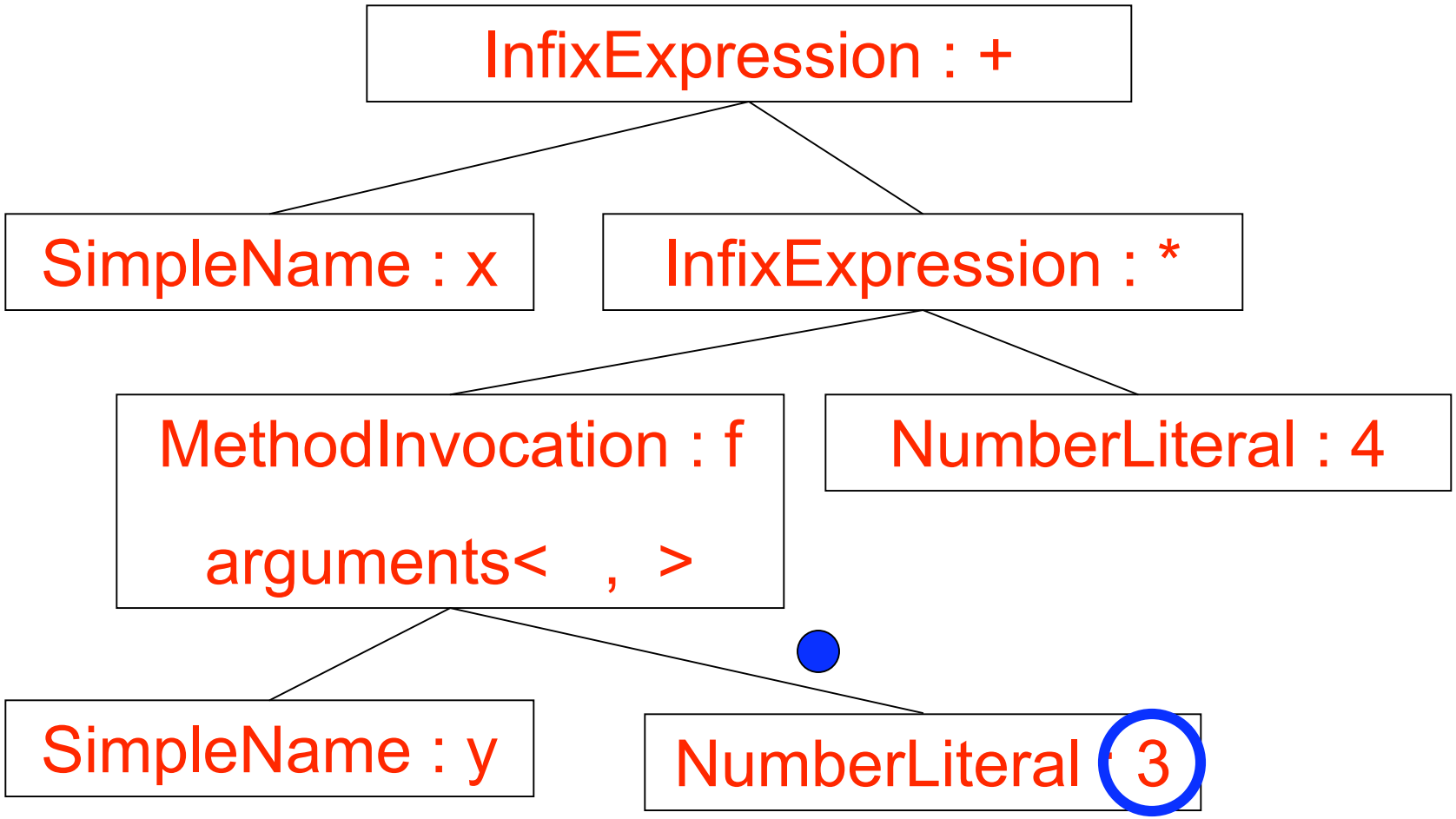
Numbers in x + f(y,3) * 4



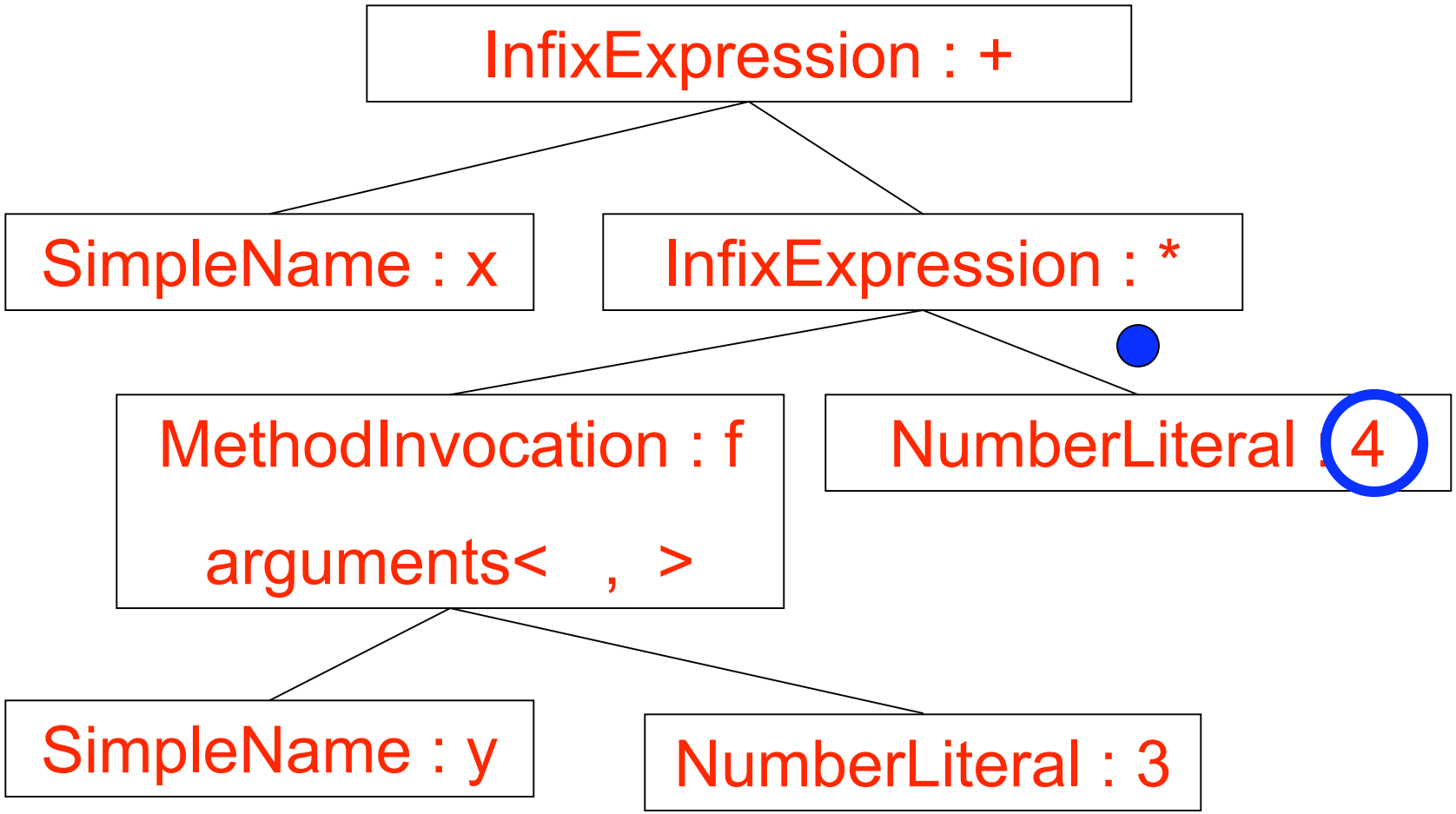
Numbers in x + f(y,3) * 4



Numbers in x + f(y,3) * 4



Numbers in x + f(y,3) * 4

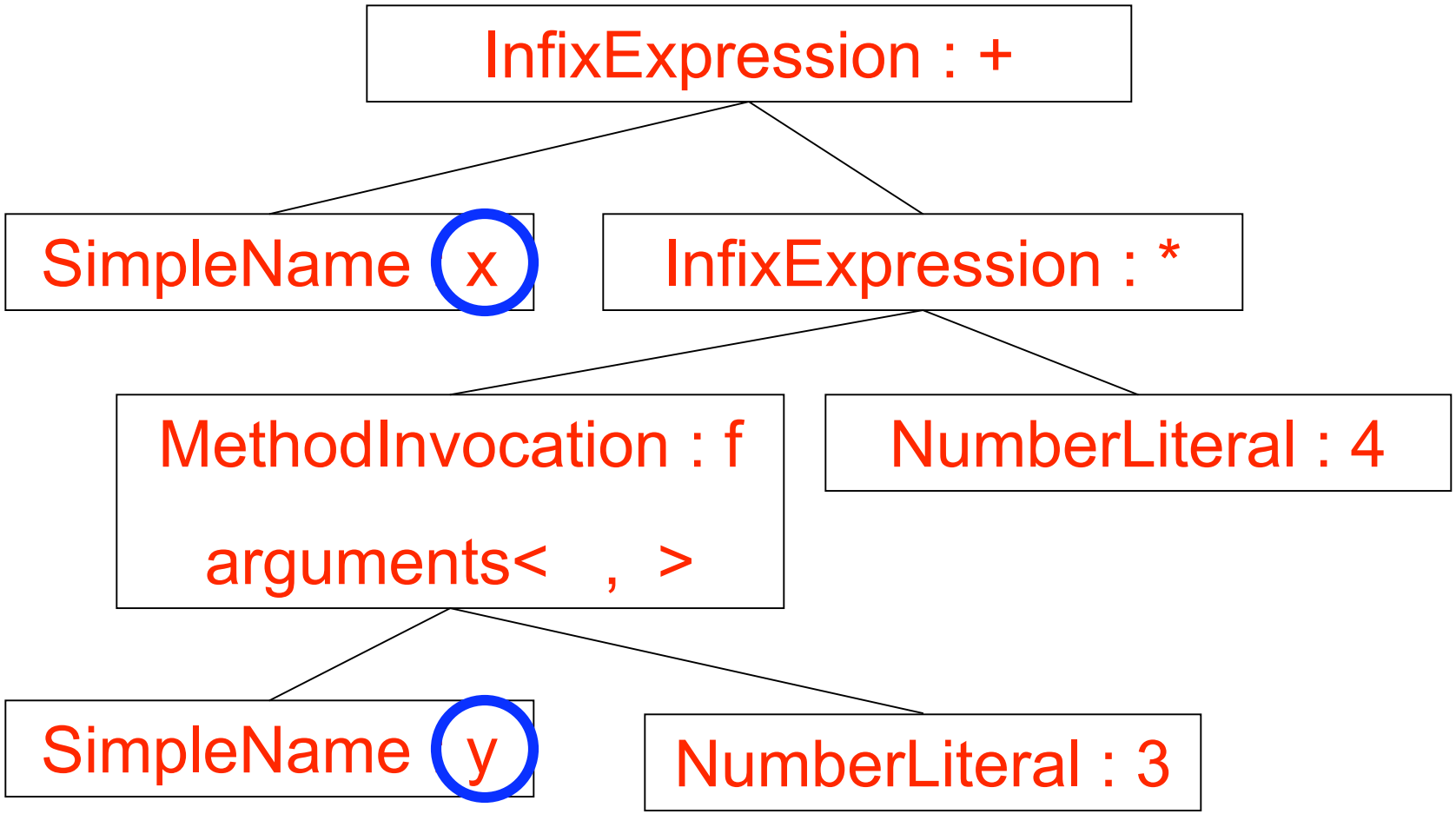


AST Traversal - Collecting Variable Names

```
public void collectName(Expression e) {
  if (e instanceof BooleanLiteral) {
    // skip
  } else if (e instanceof NumberLiteral) {
    // skip
  } else if (e instanceof PrefixExpression){
    collectName((PrefixExpression) e);
  } else if (e instanceof InfixExpression) {
    collectName((InfixExpression) e);
  } else if (e instanceof
            ParenthesizedExpression) {
    collectName((ParenthesizedExpression)e);
  } else if (e instanceof
            MethodInvocation) {
    collectName((MethodInvocation) e);
  } else if (e instanceof SimpleName) {
    collectName((SimpleName) e);
  } else { assert false; } }
```

```
public void collectName(PrefixExpression e) {
  collectName(e.getOperand());
}
public void collectName(InfixExpression e) {
  collectName(e.getLeftOperand());
  collectName(e.getRightOperand());
}
public void collectName(ParenthesizedExpression e)
{
  collectName(e.getExpression());
}
public void collectName(MethodInvocation e) {
  for (Object o : e.arguments()) {
    collectName((Expression) o);
  }
}
public void collectName(SimpleName e) {
  addName(e.getIdentifier());
}
```

Names in x + f(y,3) * 4



AST Traversal - Comparison

```
public void collectInt(Expression e) {
  if (e instanceof BooleanLiteral) {
    // skip
  } else if (e instanceof NumberLiteral) {
    collectInt((NumberLiteral) e);
  } else if (e instanceof PrefixExpression){
    collectInt((PrefixExpression) e);
  } else if (e instanceof InfixExpression) {
    collectInt((InfixExpression) e);
  } else if (e instanceof
             ParenthesizedExpression) {
    collectInt((ParenthesizedExpression) e);
  } else if (e instanceof
             MethodInvocation) {
    collectInt((MethodInvocation) e);
  } else if (e instanceof SimpleName) {
    // skip
  } else { assert false; } }
```

```
public void collectName(Expression e) {
  if (e instanceof BooleanLiteral) {
    // skip
  } else if (e instanceof NumberLiteral) {
    // skip
  } else if (e instanceof PrefixExpression){
    collectName((PrefixExpression) e);
  } else if (e instanceof InfixExpression) {
    collectName((InfixExpression) e);
  } else if (e instanceof
             ParenthesizedExpression) {
    collectName((ParenthesizedExpression)e);
  } else if (e instanceof
             MethodInvocation) {
    collectName((MethodInvocation) e);
  } else if (e instanceof SimpleName) {
    collectName((SimpleName) e);
  } else { assert false; } }
```

AST Traversal - Comparison

```
public void collectInt(NumberLiteral e) {
  addInt(Integer.parseInt(e.getToken()));
}
public void collectInt(PrefixExpression e) {
  collectInt(e.getOperand());
}
public void collectInt(InfixExpression e) {
  collectInt(e.getLeftOperand());
  collectInt(e.getRightOperand());
}
public void collectInt(ParenthesizedExpression e)
{
  collectInt(e.getExpression());
}
public void collectInt(MethodInvocation e) {
  for (Object o : e.arguments()) {
    collectInt((Expression) o);
  }
}
```

```
public void collectName(PrefixExpression e) {
  collectName(e.getOperand());
}
public void collectName(SimpleName e) {
  addName(e.getIdentifier());
}
void collectName(InfixExpression e) {
  collectName(e.getLeftOperand());
  collectName(e.getRightOperand());
}
public void collectName(ParenthesizedExpression e)
{
  collectName(e.getExpression());
}
public void collectName(MethodInvocation e) {
  for (Object o : e.arguments()) {
    collectName((Expression) o);
  }
}
```

Assessment (1)

- Each collecting algorithm requires
  - The traversal of the sub-structure of expression, e.g., collect*X* (Expression)
  - The part-specific processing, i.e., collectInt(NumberLiteral) and collectName(SimpleName)
- Traversing the structure of the data is common to all of the operations
  - but it is implemented independently in each algorithm

Assessment (2)

- What happens if we add another kind of expression in our AST/language?
  - assertion failure happens (if assertion check is enabled, otherwise ?)
  - need to add a new if-else case for both code base
    - what if we have various algorithms that work on the AST?
    - we need to adapt all the (traversal) code
- Can we find a way to factor traversal out of an operation?
  - we still need to be able to specify the part-specific processing

AST Traversal - Visitor Pattern in JDT

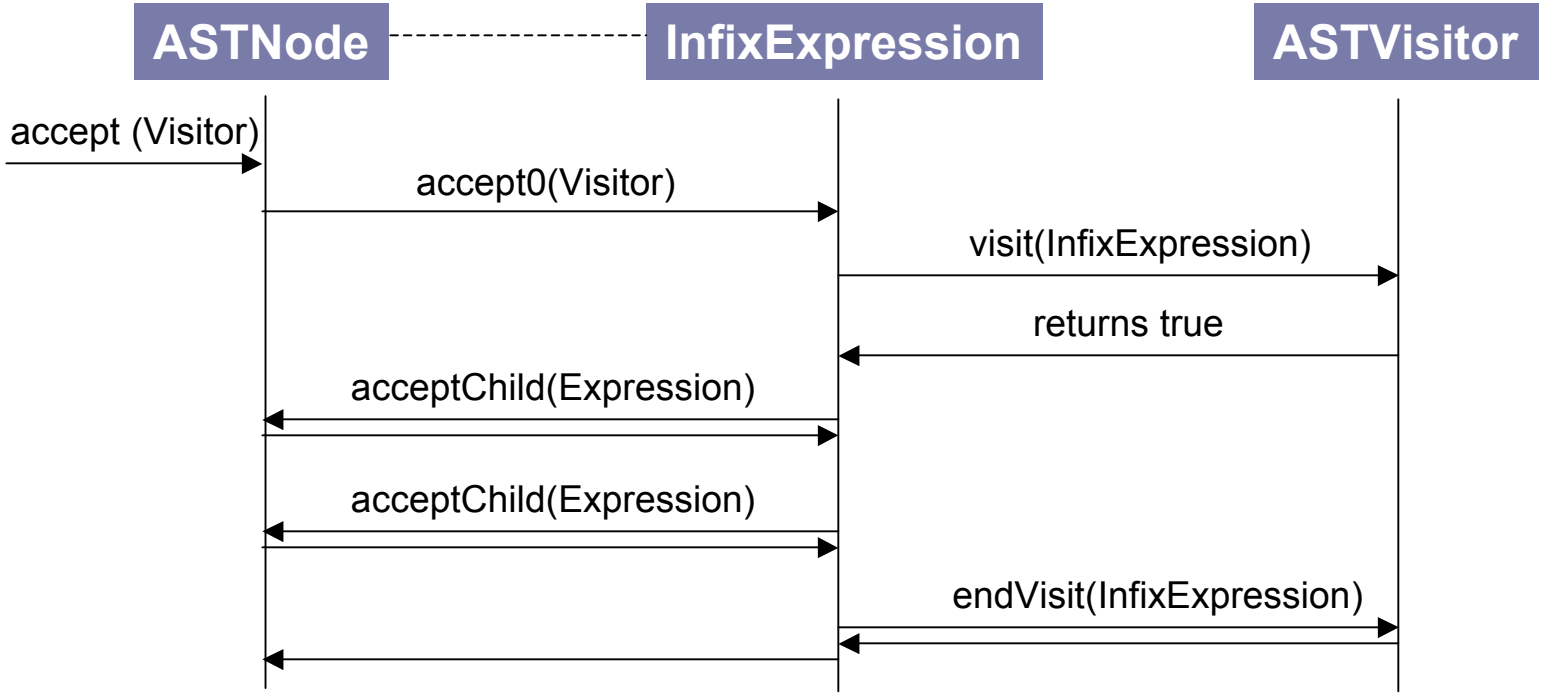
```
public abstract class ASTNode {
  abstract void accept0(ASTVisitor visitor);
  public void accept(ASTVisitor visitor) {
    if (visitor == null) {
      throw new IllegalArgumentException();
    }
    …
    accept0(visitor);
    …
  }
  final void acceptChild(ASTVisitor visitor,
                         ASTNode child) {
    if (child == null) { return; }
    child.accept(visitor);
  }
}
```

```
public class InfixExpression extends Expression {
  void accept0(ASTVisitor visitor) {
    boolean visitChildren = visitor.visit(this);
    if (visitChildren) {
      acceptChild(visitor, getLeftOperand());
      acceptChild(visitor, getRightOperand());
      …
    }
    visitor.endVisit(this);
  }
}

public abstract class ASTVisitor {
  public boolean visit(InfixExpression node) {
    return true;
  }
  public void endVisit(InfixExpression node) {}
}
```

Class Interaction

- Suppose we have an InfixExpression e, when we invoke e.accept(visitor)



Assessment (4)

- ASTNode provides a generic accept method to initiate AST traversal
- Each AST node (e.g., InfixExpression) implements the abstract accept0 method that traverse its subnodes
- ASTVisitor provides default methods that basically do nothing except instructing that the node traversal should continue to visit subnodes

Assessment (5)

- Notice that if-else and type casting to handle different kinds of AST node are not needed
  - replaced by dynamic dispatch of accept0
- When one overrides the ASTVisitor visit and endVisit methods
  - dynamic method dispatch will not call the default methods in ASTVisitor
  - it will invoke the overriding methods

AST Traversal - Collecting Int Literal Revisited

```
public abstract class ASTNode {
  abstract void accept0(ASTVisitor visitor);
  public void accept(ASTVisitor visitor) {
    if (visitor == null) {
      throw new IllegalArgumentException();
    }
    …
    accept0(visitor);
    …
  }
  final void acceptChild(ASTVisitor visitor,
                         ASTNode child) {
    if (child == null) { return; }
    child.accept(visitor);
  }
}
```

```
public class NumberLiteral extends Expression {
  void accept0(ASTVisitor visitor) {
    visitor.visit(this);
    visitor.endVisit(this);
  }
}
public abstract class ASTVisitor {
  public boolean visit(NumberLiteral node) {
    return true;
  }
  public void endVisit(NumberLiteral node) {}
}
…
e.accept(new ASTVisitor() {
  public boolean visit(NumberLiteral node) {
    addInt(Integer.parseInt(node.getToken()));
    return false;
  }
});
```

AST Traversal - Collecting Var Name Revisited

```
public abstract class ASTNode {
  abstract void accept0(ASTVisitor visitor);
  public void accept(ASTVisitor visitor) {
    if (visitor == null) {
      throw new IllegalArgumentException();
    }
    …
    accept0(visitor);
    …
  }
  final void acceptChild(ASTVisitor visitor,
                         ASTNode child) {
    if (child == null) { return; }
    child.accept(visitor);
  }
}
```

```
public class SimpleName extends Expression {
  void accept0(ASTVisitor visitor) {
    visitor.visit(this);
    visitor.endVisit(this);
  }
}
public abstract class ASTVisitor {
  public boolean visit(SimpleName node) {
    return true;
  }
  public void endVisit(SimpleName node) {}
}
…
e.accept(new ASTVisitor() {
  public boolean visit(SimpleName node) {
    addName(node.getIdentifier());
    return false;
  }
});
```

For You To Do

- Does the code in the previous slide only collect variable names?
  - look at the various Expression API in Eclipse JDT DOM and think about what SimpleName are used for
- Fix the previous code accordingly
  - hint: see the code for `collectName(MethodInvocation e)`

Assessment (6)

- What happens if we add another kind of expression in our AST/language?
  - the new node is responsible for its traversal (changes are localized)
  - need to add default methods in the AST visitor
- Avoiding pitfalls
  - notice that if we add another kind of node, then our previous code may not break (as in the assertion failure case)
    - however, we still need to make sure that the new node does not require changes in the actual processing algorithm
  - we need to keep in mind the context of where the AST node can be traversed from

Weeding-out Divide by Zero

- In the grammar, we can prevent the following input: “3 / 0”
  - we can use ANTLR semantic predicates in the grammar
  - but, we can do a separate check pass

```
cu.accept(new ASTVisitor() {
   public boolean visit(InfixExpression node) {
     if (node.getOperator() == InfixExpression.Operator.DIVIDE) {
       if ("0".equals(node.getRightOperand().toString().trim())) {
         // signal error
       }
     }
     return super.visit(node);
   } });
```

Weeding-out Multi Array Access

- In the grammar, we can prevent the following input: “a[0][0]”
  - we can use ANTLR semantic predicates in the grammar
  - but, we can do a separate check pass

```
cu.accept(new ASTVisitor() {
    public boolean visit(ArrayAccess node) {
      if (node.getArray() instanceof ArrayAccess) {
        // signal error
      }
      return super.visit(node);
    }
});
```