COMP3131/9102: Programming Languages and Compilers

Jingling Xue

School of Computer Science and Engineering
The University of New South Wales
Sydney, NSW 2052, Australia
`http://www.cse.unsw.edu.au/~cs3131`
`http://www.cse.unsw.edu.au/~cs9102`

Week 4 (1st & 2nd Lectures): Attribute Grammars

1st Lec

1. Attribute grammars (D. E. Knuth (1968))
   - S-attributed grammars
   - L-attributed grammars
2. Attributes (synthesised and inherited)
3. Semantic rules (or functions)
4. The computation of attributes
   - tree walkers in one or multiple passes – evaluation order determined at compile time
   - rule-based – evaluation order fixed at compiler-construction time
     - can be used in the presence of a tree
     - parsing and checking in one-pass without using a tree
5. Visitor design pattern
6. Assignment 3

2nd Lec

Examples

- Example 1:
  - Attribute grammars
  - Synthesised and inherited attributes
  - Attribute evaluation
- Example 2: L-attributed
- Example 3: S-attributed (revisited from Week 3 (2nd lecture))

1st Lec

2nd Lec

Rule-Based Method for Example 1: the Main Method

- Slides 284 – Slide 290: writing a parser
- Slides 291 – Slide 297: writing a rule-based evaluator using the visitor design pattern

An Unambiguous Grammar for Parsing

- Left-recursive:

$$\begin{array}{rcl}\langle \text{S} \rangle & \to & \langle \text{expr} \rangle \\ \langle \text{expr} \rangle & \to & \langle \text{expr} \rangle \ \textbf{/} \ \langle \text{factor} \rangle \\ \langle \text{expr} \rangle & \to & \langle \text{factor} \rangle \\ \langle \text{factor} \rangle & \to & \textbf{num} \\ \langle \text{factor} \rangle & \to & \textbf{num . num}\end{array}$$

- Non-left-recursive for top-down parsing:

$$\begin{array}{rcl}\langle \text{S} \rangle & \to & \langle \text{expr} \rangle \\ \langle \text{expr} \rangle & \to & \langle \text{factor} \rangle \ (\textbf{/} \ \langle \text{factor} \rangle)^* \\ \langle \text{factor} \rangle & \to & \textbf{num} \\ \langle \text{factor} \rangle & \to & \textbf{num . num}\end{array}$$

AST Classes Used for Building ASTs for Example 1

```
+ AST
  = S
  + Expr
    = BinaryExpr
    = IntExpr
    = FloatExpr
  + Terminal
    = IntLiteral
    = FloatLiteral
    = Operator
```

These are the same as those in VC.ASTs except S and Expr.

S.java

```
package VC.ASTs;

import VC.Scanner.SourcePosition;

public class S extends AST {
  public Expr E;
  public String val;

  public S(Expr eAST, SourcePosition position) {
    super (position);
    E = eAST;
  }
  public Object visit(Visitor v, Object o) {
    return v.visitS(this, o);
  }
}
```

Expr.java

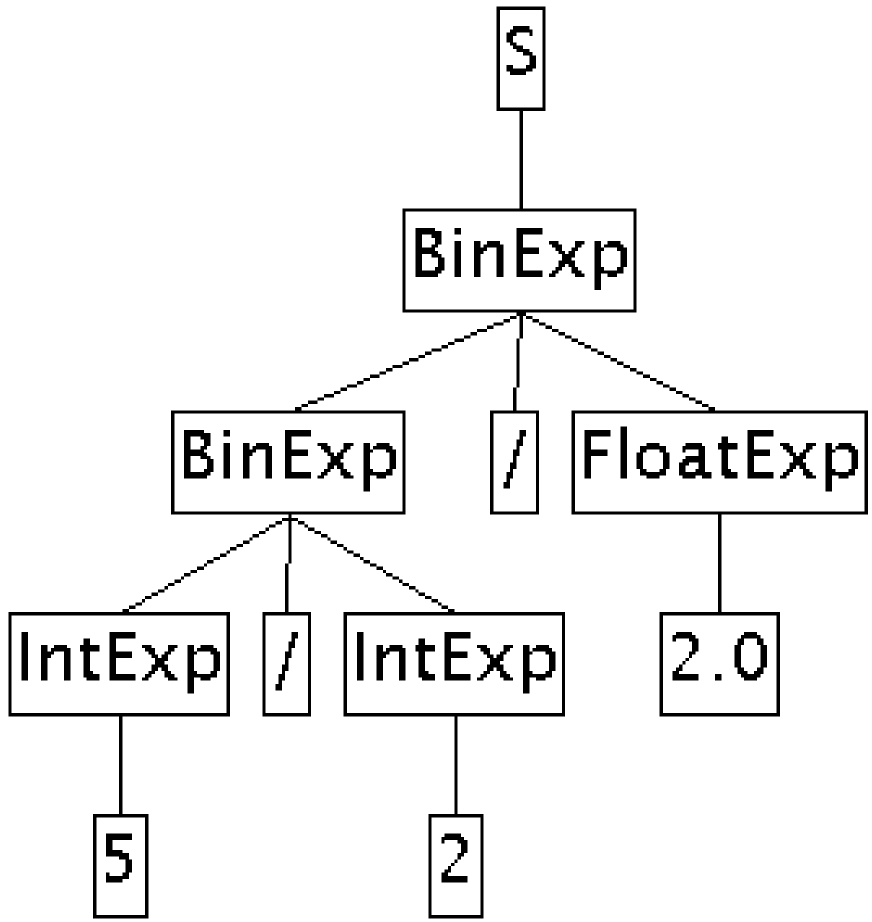
```
package VC.ASTs;

import VC.Scanner.SourcePosition;

public abstract class Expr extends AST {
  // attributes
  public boolean isFloat;
  public String type;
  public String val;

  public Expr (SourcePosition Position) {
    super (Position);
    type = null;
  }
}
```

AST-Building Parser (Written as Per Week 3 (2nd Lecture))

```
public class Parser {
  private Scanner scanner;
  private ErrorReporter errorReporter;
  private Token currentToken;

  public Parser (Scanner lexer, ErrorReporter reporter) {
    scanner = lexer;
    errorReporter = reporter;
    currentToken = scanner.getToken();
  }

  void accept() {
    currentToken = scanner.getToken();
  }

  void syntacticError(String messageTemplate, String tokenQuoted) {
    SourcePosition pos = currentToken.position;
    errorReporter.reportError(messageTemplate, tokenQuoted, pos);
  }

  public S parseS() {
    S sAST = null;
    SourcePosition dummyPos = new SourcePosition();

    try {
      Expr eAST = parseExpr();
      sAST = new S(eAST, dummyPos);
      if (currentToken.kind != Token.EOF)
        syntacticError("\"%\" invalid expression", currentToken.spelling);
    } catch (SyntaxError e) { return null; }
    return sAST;
  }

  Expr parseExpr() throws SyntaxError {
    Expr exprAST = null;
    SourcePosition dummyPos = new SourcePosition();
```

```
    exprAST = parseFactor();
    while (currentToken.kind == Token.DIV) {
      Operator opAST = new Operator(currentToken.spelling, dummyPos);
      accept();
      Expr e2AST = parseFactor();
      exprAST = new BinaryExpr(exprAST, opAST, e2AST, dummyPos);
    }
    return exprAST;
  }

  Expr parseFactor() throws SyntaxError {
    Expr eAST = null;
    SourcePosition dummyPos = new SourcePosition();

    switch (currentToken.kind) {
    case Token.INTLITERAL:
        eAST = new IntExpr(new IntLiteral(currentToken.spelling, dummyPos), dummyPos);
        accept();
        break;
    case Token.FLOATLITERAL:
        eAST = new FloatExpr(new FloatLiteral(currentToken.spelling, dummyPos), dummyPos);
        accept();
        break;
    default:
        syntacticError("\"%\" cannot start Factor", currentToken.spelling);
        break;
    }
   return eAST;
  }
}
```

The AST for **5/2/2.0**



Visitor.java

```
/*
 * Visitor.java
 */

package VC.ASTs;

public interface Visitor {

  public abstract Object visitS(S ast, Object o);
  public abstract Object visitBinaryExpr(BinaryExpr ast, Object o);
  public abstract Object visitIntExpr(IntExpr ast, Object o);
  public abstract Object visitFloatExpr(FloatExpr ast, Object o);
  public abstract Object visitIntLiteral(IntLiteral ast, Object o);
  public abstract Object visitFloatLiteral(FloatLiteral ast, Object o);
  public abstract Object visitOperator(Operator ast, Object o);

}
```

- o: pass some inherited attributes top-down
- returned object: pass synthesised attributes bottom-up

The Visitor Design Pattern

```
public class BinaryExpr extends Expr { // for defining BinaryExpr nodes
  ...
  public Object visit(Visitor v, Object o) {
    return v.visitBinaryExpr(this, o); // this: the node
  }
}
public class IntExpr extends Expr { // for defining IntExpr nodes
  ...
  public Object visit(Visitor v, Object o) {
    return v.visitIntExpr(this, o); // this: the node
  }
}
public xyzVisitor implements Visitor {
  Object visitBinaryExpr(BinaryExpr ast, Object o) {
    ast.E1.visit(this, o) // this: the visitor
    ast.E2.visit(this, o)
    // do some thing on this BinaryExpr node "ast"
    return something
  }
  ...
}
```

The Visitor Design Pattern

- The visitor contains operations that operate on data defined by other classes (e.g., the nodes in the AST)
- Can design different visitors that do different things on the same data
- Simple to implement
- Useful for writing an interpreter, type checker, code generator, ...
- Less useful at development stage since one needs to add a new abstract class to the visitor interface every time when one adds new classes (say, for new AST nodes) which must be visited
- `http://en.wikipedia.org/wiki/Visitor_pattern`
  `https://dzone.com/articles/visitor-design-pattern-in-java`
  `http://sourcemaking.com/design_patterns/visitor/java/1`

Rule-Based Method for Example 1

```
...
 theAST = parser.parseS();

 visitor1 = new FloatVisitor();
 visitor1.evalFloat(theAST, null);
 visitor2 = new TypeValVisitor();
 visitor2.evalTypeVal(theAST, null);
 System.out.println("The value is: " + theAST.val);
...
```

Pass 1: Computing isFloat (Post-Order Traversal)

```
// FloatVisitor.java -- o and returned results not used
import VC.ASTs.*;
public class FloatVisitor implements Visitor {
  public void evalFloat(S ast, Object o) {
    ast.visit(this, o);
  }
  public Object visitS(S ast, Object o) {
    ast.E.visit(this, o);
    return null;
  }
  public Object visitBinaryExpr(BinaryExpr ast, Object o) {
    ast.E1.visit(this, o);
    ast.E2.visit(this, o);
    ast.isFloat = ast.E1.isFloat || ast.E2.isFloat;
    return null;
  }
  public Object visitIntExpr(IntExpr ast, Object o) {
    ast.isFloat = false;
    return null;
  }
  public Object visitFloatExpr(FloatExpr ast, Object o) {
    ast.isFloat = true;
    return null;
  }
  // not called
```

```
  public Object visitIntLiteral(IntLiteral ast, Object o) {
    return null;
  }
  // not called
  public Object visitFloatLiteral(FloatLiteral ast, Object o) {
    return null;
  }
  // not called
  public Object visitOperator(Operator ast, Object o) {
    return null;
  }
}
```

By using the visitor pattern, each visit method at a node knows which particular visitor method to call. For example, IntExpr's visit method knows that this is an IntExpr object and calls v.visitIntExpr, passing this and o as arguments.

Pass 2: Computing type (Pre-order) and val (Post-Order) – Combined

```
// TypeValVisitor.java -- o and returned result not used.
import VC.ASTs.*;
public class TypeValVisitor implements Visitor {
  public void evalTypeVal(S ast, Object o) {
    ast.visit(this, o);
  }
  public Object visitS(S ast, Object o) {
    if (ast.E.isFloat)
      ast.E.type = "float";
    else
      ast.E.type = "int";
    ast.E.visit(this, o);
    ast.val = ast.E.val;
    return null;
  }
  public Object visitBinaryExpr(BinaryExpr ast, Object o) {
    ast.E1.type = ast.type;
    ast.E2.type = ast.type;
    ast.E1.visit(this, o);
    ast.E2.visit(this, o);
    // "/" in Java overloaded for integer and floating-point divisions
    if (ast.type.equals("float"))
        ast.val = String.valueOf(Float.parseFloat(ast.E1.val) /
            Float.parseFloat(ast.E2.val));
    else
        ast.val = String.valueOf(Integer.parseInt(ast.E1.val) /
            Integer.parseInt(ast.E2.val));
    return null;
  }
  public Object visitIntExpr(IntExpr ast, Object o) {
```

```
    String num = (String) ast.IL.visit(this, o);
    if (ast.type.equals("float"))
      ast.val = String.valueOf((float) Integer.parseInt(num));
    else
      ast.val = num;
    return null;
  }
  public Object visitFloatExpr(FloatExpr ast, Object o) {
    ast.val = (String) ast.FL.visit(this, o);
    return null;
  }
  public Object visitIntLiteral(IntLiteral ast, Object o) {
    return ast.spelling;
  }
  public Object visitFloatLiteral(FloatLiteral ast, Object o) {
    return ast.spelling;
  }
  // not called
  public Object visitOperator(Operator ast, Object o) { return null; }
}
```

L-Attributed Grammars

- Motivation: parsing and semantic analysis in one pass in top-down parsers (recursive descent and LL parsers)
- Definition: An attribute grammar is L-attributed if each inherited attribute of X_i, $1 \leqslant i \leqslant n$, on the right-hand side of $X_0 \rightarrow X_1 X_2 \cdots X_m$, depends only on:
  - the attributes of the symbols $X_1, X_2, \cdots, X_{i-1}$ to the left of X_i in the production, and
  - the inherited attributes of X_0
- The L: the information flowing from left to right
- Example 1 grammar is not L-attributed (because in $\langle$S $\rightarrow$expr$\rangle$, the inherited attribute $\langle$expr.type$\rangle$ depends on the synthesised attribute $\langle$expr.isFloat$\rangle$)

Evaluation of L-attributed Grammars

```
void dfvisit (AST N) {
  for ( each child M of N from left to right ) {
    evaluate inherited attributes of M;
    dfvisit(m);
  }
  evaluate synthesised attributes of N
}
```

- All attributes can be evaluated in one pass
- Parsing and semantic analysis can be done together (say, in a recursive descent parser) without using a tree

Example 2: L-Attributed Grammar

- The grammar (EBNF):

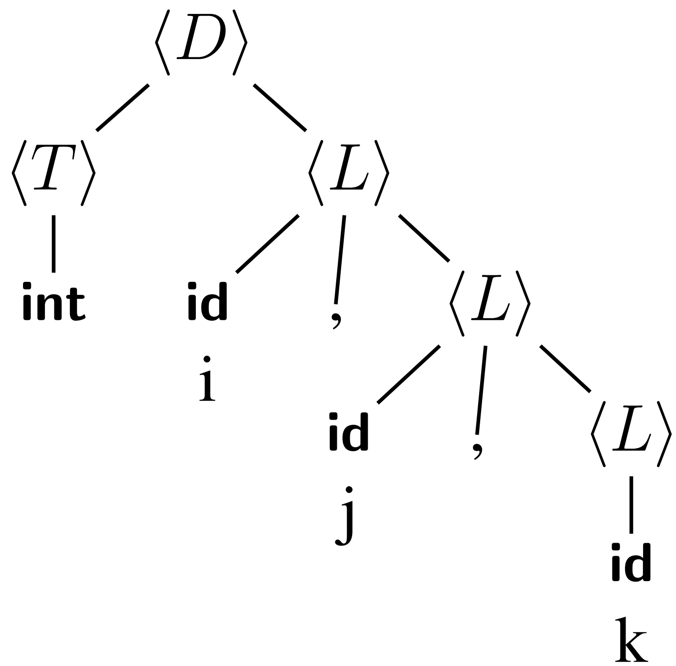
$$
\begin{array}{lcl}
D & \rightarrow & T\ \textbf{id}(,\textbf{id})^* \\
T & \rightarrow & \textbf{int} \\
T & \rightarrow & \textbf{float}
\end{array}
$$

- The grammar (BNF):

$$
\begin{array}{lcl}
D & \rightarrow & T\,L \\
T & \rightarrow & \textbf{int} \\
T & \rightarrow & \textbf{float} \\
L & \rightarrow & \textbf{id}, L \\
L & \rightarrow & \textbf{id}
\end{array}
$$

- Give an attribute grammar for constructing the AST.

Parse Tree for int i, j, k using the BNF Grammar



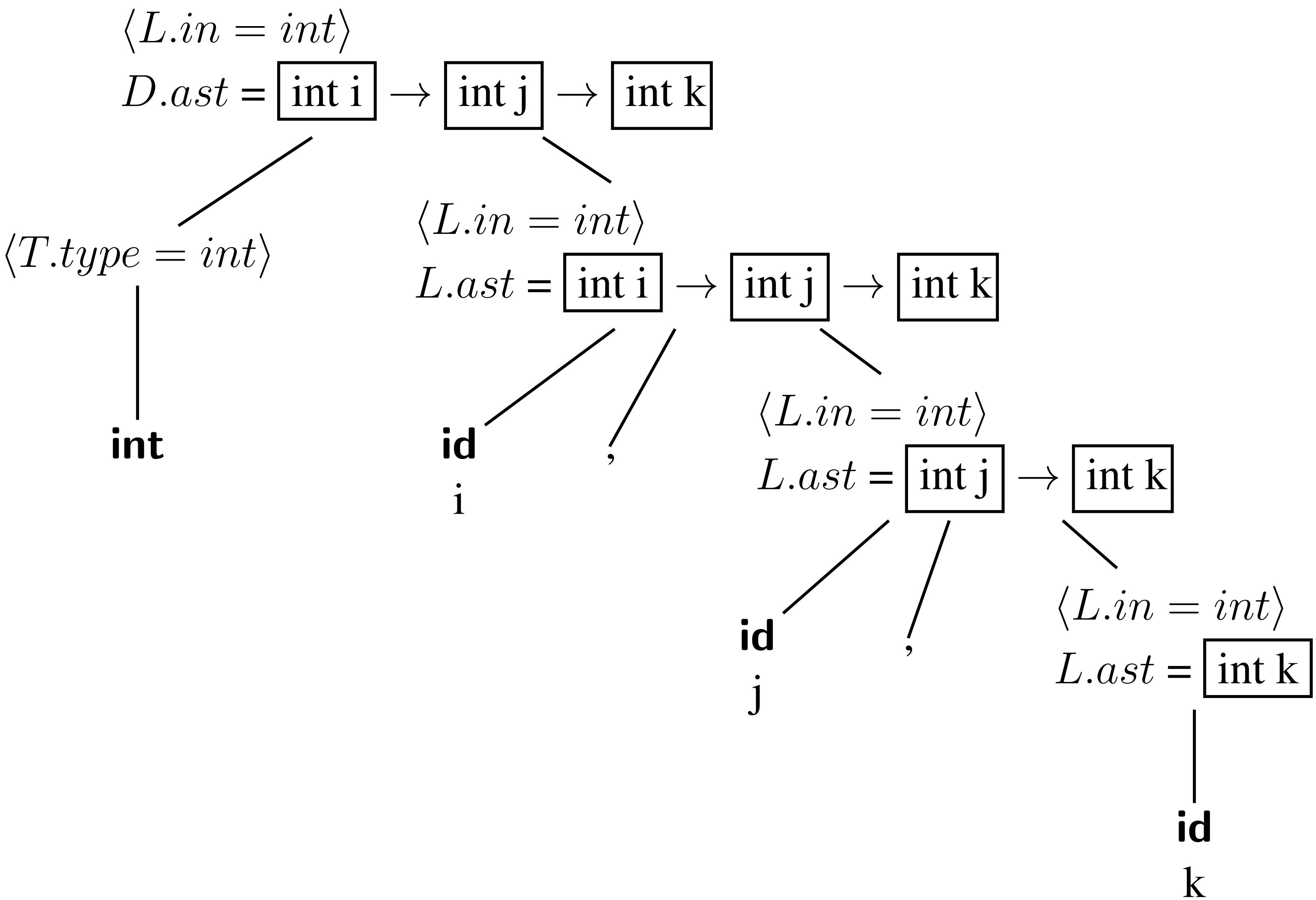
An Attribute Grammar for Example 2

Production	Semantic Rules
$D \quad \to \quad T L$	$L.\langle\text{in}\rangle = T.\langle\text{type}\rangle$
	$D.\langle\text{ast}\rangle = L.\langle\text{ast}\rangle$
$T \quad \to \quad$ **int**	$T.\langle\text{type = int}\rangle$
$T \quad \to \quad$ **float**	$T.\langle\text{type = float}\rangle$
$L \quad \to \quad$ **id**, L_1	$L_1.\langle\text{in}\rangle = L.\langle\text{in}\rangle$
	L.ast = new DeclList(new VarDecl(L.in, ”id”), L_1.ast)
$L \quad \to \quad$ **id**	L.ast = new DeclList(new VarDecl(L.in, ”id”), null)

- type: synthesised attribute
- ast: synthesised attribute
- in: inherited attribute

Relevant to Assignment 3

Decorated Parse Tree



S-Attributed Grammars

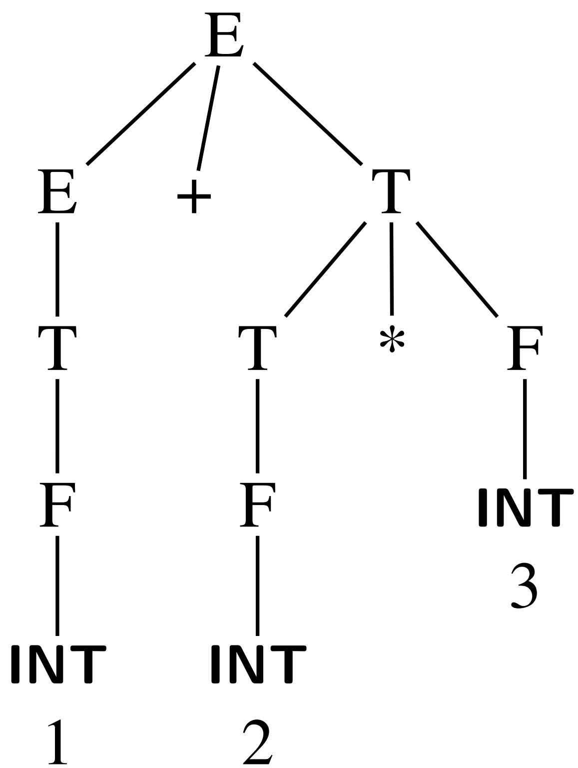
- Motivation: parsing and semantic analysis in one pass in bottom-up parsers
- Definition: An attribute grammar is S-attributed if it uses synthesised attributes only
- The information always flow up in the tree
- Every S-attributed grammar is L-attributed
- Examples 1 and 2 are not S-attributed

Example 3: S-Attributed Grammar (Slide 268)

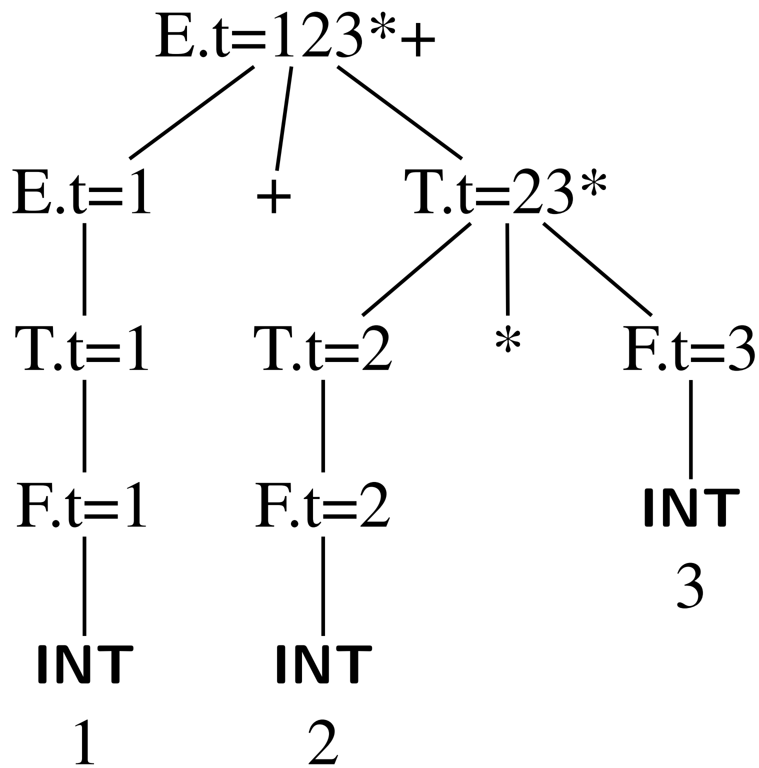
PRODUCTION	SEMANTIC RULE
$E \to T$	$[E.t = T.t]$
$\mid\ E_1$"+"$\ T$	$[E.t = E_1.t \parallel T.t \parallel$ "+"$]$
$\mid\ E_1$ "-"$\ T$	$[E.t = E_1.t \parallel T.t \parallel$ "-"$]$
$T \to F$	$[T.t = F.t]$
$\mid\ T_1$" $*$ "F	$[T.t = T_1.t \parallel F.t \parallel$ "*"$]$
$\mid\ T_1$"/"$\ F$	$[T.t = T_1.t \parallel F.t \parallel$ "/"$]$
$F \to$ **INT**	$[F.t =$ **int**.string-value$]$
$F \to$ "(" E ")"	$[F.t = E.t]$

- Defines the translation of infix to postfix expressions
- A single string-valued synthesised attribute t
- $\parallel$: string concatenation

Example 3: Understanding Example 3



Parse Tree



Decorated Parse Tree

- The value of the synthesised attribute t is propagated upwards the tree

What You Should Know About Attribute Grammars?

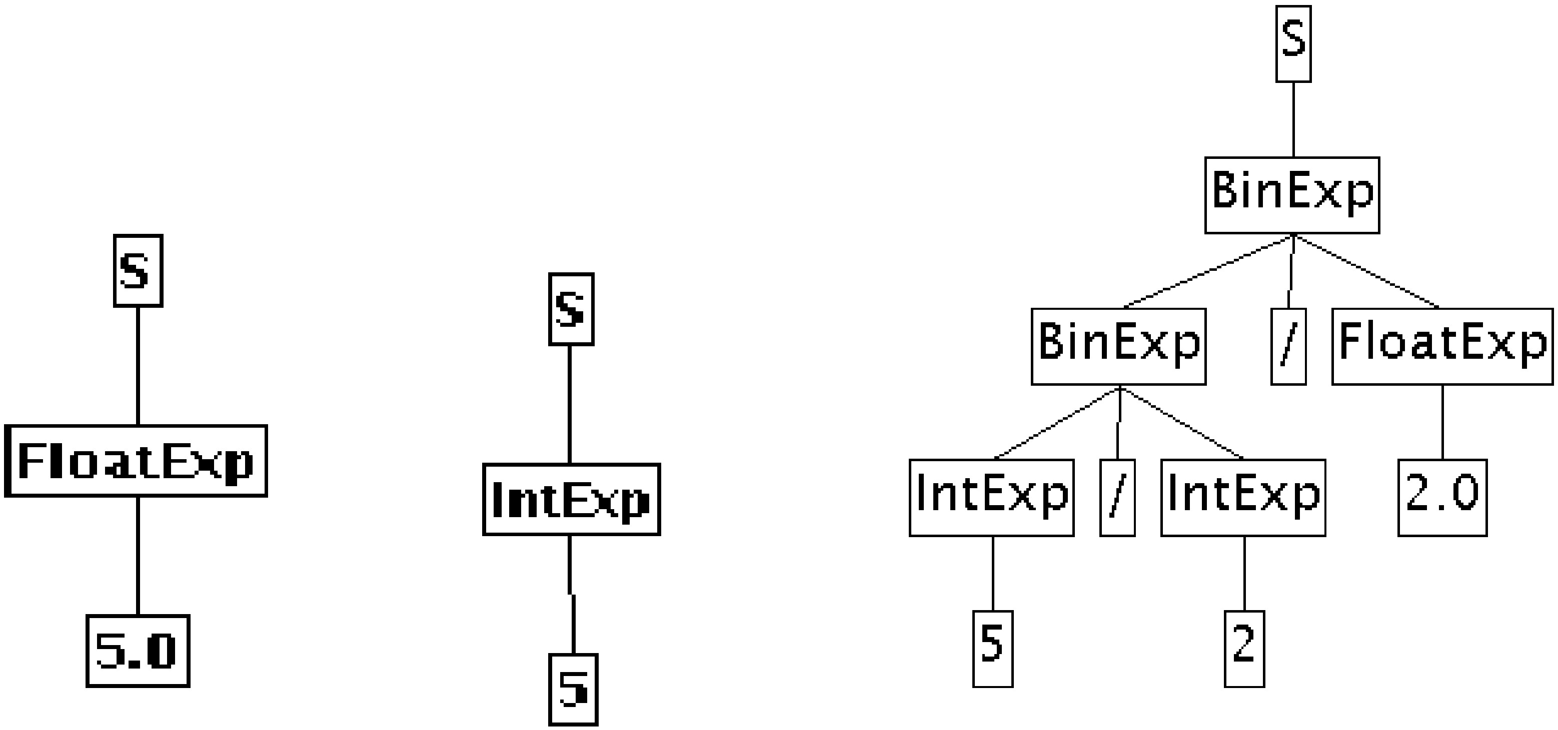
- Write an attribute grammar for simple CFGs
- Apply Slide **??** to evaluate an attribute grammar
- Draw decorated parse or decorated syntax trees
- Evaluate the attributes

Reading

- Attribute grammars: Sections 2.3 and 5.1 – 5.4 (either version)
- Understand the visitor design pattern:
  - Read Slide 293
  - Read TreeDrawer, TreePrinter and UnParser

Next Week: Static Semantics

Different Types of S's Child



Different Types of the Left Child of a BinExpr Node